PhD in Computer Science and Engineering

Cycle XX

University of Modena and Reggio Emilia

Dissertation thesis for the PhD in Computer Science and Engineering

The Issue of Strong Mobility: an Innovative Approach based on the IBM Jikes Research Virtual Machine

Candidate:
Ing. Raffaele Quitadamo

Sign:
_____

Supervisor:
Prof. Letizia Leonardi

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Physical mobility of portable devices and personal mobility are changing the everyday life of many people around the world. A less evident, but equally revolutionary, form of mobility has been reshaping the *logical* structure of distributed systems in the last few years. **Computational mobility**, also known as **mobile code**, introduces the idea of changing at runtime the machine where a piece of software executes during its life. A number of researchers both in academia and industry have acknowledged that *mobile components* (i.e. software components able to move from one node to another on need) can significantly overcome the fragility and lack of flexibility of traditional client-server interactions when occurring in a large-scale distributed system.

Although a significant number of languages have been created or simply adapted to implement mobile components, the key role played by the Java language in the last decade is beyond dispute. The **Java Virtual Machine (JVM)** has always been chosen as the best execution environment for mobile components, mostly because of its portable intermediate language (known as *bytecode*) and the JDK built-in support to object persistence (called *Object Serialization*). The computational state of a Java application can be decomposed in its bytecode (code segment), the set of referenced objects in the heap (dataspace) and the call stack with context registers (execution state). While it is pretty straightforward to move the bytecode (e.g. using url classloaders) and the referenced objects among different JVMs (using serialization), there is no such a standard way to move the call stack. This has led to the adoption of a "weaker form of mobility" in many scenarios (e.g. mobile agents in e-commerce applications): **weak mobility** means moving only the code segment and the dataspace, disrupting the call stack and thus causing the component to restart from the beginning (or from predefined checkpoints) at each movement.

Yet, in distributed and parallel computing applications keeping the execution state of a computation is of paramount importance to perform load-balancing or to achieve fault-tolerance and persistence. The kind of mobility that allows a full migration of the computational state is known as **strong mobility** and is the subject defended by the present dissertation thesis. How to easily and efficiently relocate the execution state of a Java thread is still a debated topic in the mobile code community. Java threads are objects with a deep and strong binding to the underlying operating system and, for efficiency and security reasons, user programs can access only a limited and safe set of methods to manage them. Structures like the call stack and machine registers are visible only to the JVM runtime and cannot be extracted or modified from the outside.

The problem of thread migration in Java has been dealt with from dif-

ferent points of view: some approaches chose to manipulate or instrument the bytecode to capture thread state in a portable manner, while others rely on modifications of the JVM runtime. The former systems are, on the one hand, characterized by overwhelming overheads on the execution time of threads (88% to 250%). The latter systems, on the other hand, often use old interpreter-based JVMs, thus heavily limiting performance of the applications.

The present dissertation thesis reports on a novel thread migration approach in Java that overcomes the limitations of previous systems. The whole approach has been implemented on top of the IBM Jikes Research Virtual Machine, developed at the T. J. Watson Labs in New York and now donated by IBM to the open source community. **Mobile JikesRVM**, the Java framework that realizes the proposed approach, provides the programmer with features like:

- Fast and complete thread state capture and restoration.
- Minimum overhead on the execution of the compiled code.
- Efficient integration in a JIT-compiled JVM, even in presence of aggressive code optimizations.
- Concise and simple API to migrate and restore threads.
- Full support to both practive (i.e. spontaneous) or reactive (triggered by another thread) migration.

The remainder of this thesis is organized as follows:

- Chapter 2 introduces basic concepts on code mobility, along with a classification of software engineering paradigms.
- Chapter 3 focuses on thread migration in Java, describing the JVM runtime structures involved and then surveying approaches in the literature.
- Chapter 4 presents to the reader the IBM JikesRVM project on which the proposed approach has been implemented. For the sake of simplicity and brevity, the chapter describes only features deemed strategic for the overall understanding of the implemented approach.
- Chapter 5 is the core of the entire thesis. It describes the taken design choices and the techniques implemented to achieve strong thread migration.
- The Mobile JikesRVM framework has been thoroughly tested with the JVM DaCapo benchmarks suite and its performance evaluation is the subject of Chapter 6.

- Chapter 7 reports on the application of Mobile JikesRVM to two real distributed computing scenarios, namely robot coordination with the PIM and the Agile Computing service-oriented architecture. The work in this chapter is the outcome of a 6 month research visit spent by the student at the Institute for Human and Machine Cognition (IHMC) in Pensacola, Florida, USA.

- Chapter 8 concludes the thesis, summing up the achieved research results.

Chapter 2

An overview of code mobility

2.1 Code mobility

Distributed systems [77] in the last decades have certainly proven to offer a serious platform for stable, long-lived and flexible applications development. In the last years, a great deal of research has focused on the exploitation of new broadband communication devices and in the provision of new services on large scale distributed systems, such as the Internet. More recently, the increasing pervasiveness of cellular phones and wireless LANs has added a significant degree of *mobility* in software systems and networks. The very *mobility* concept has in fact been used to refer to some well known situations like:

- **Personal Mobility**, where users may want to move from one location to another without carrying around any physical pieces of hardware. It should be possible for them to continue their work regardless of the machine they are assigned to or the place where they are.
- **Computer Mobility**, thanks to whom a growing number of useful applications can be executed on relatively cheap portable computers, such as PDAs and smartphones.

Besides those two popular forms of *physical mobility* (i.e. users or hardware move), a less evident but equally revolutionary form of mobility has been reshaping the *logical* structure of distributed systems. **Computational mobility**, also known as **mobile code**, introduces the idea of changing the machine where a piece of software executes. It has been commonly defined [34] as

> "*the capability to reconfigure dynamically, at runtime, the binding between the software components of an application and their physical location within a computer network*".

Work on code mobility is not new; it has a long and varied history, beginning with computing pioneer John Von Neumann's seminal concept of one automaton controlling another. In the 1960s, the mobile code idea was evident in remote job-entry terminals that transferred programs to mainframe computers. Lately, in the 1990s, mobile code languages started to emerge from both industry and academia. As it happens often with computer science technologies, marketing forces played a fundamental role in the acceptance of two languages coming from industry: Telescript (by General Magic [36]) and, more importantly, Java by Sun Microsystem [71] were introduced, marking the first widely used mobile code implementations.

The key conceptual contribution of mobile code has been to raise the location where an application component is executed from the status of

configuration or deployment detail to that of first-class element in the application design. Conventional distributed systems typically assume a static configuration of the environment where the distributed application executes. Communication among a set of nodes is enabled by physical links whose configuration is fixed and statically determined. Similarly, the various components of the distributed application running on the nodes of the system are bound to such nodes for their whole life. A simple architectural view of a traditional distributed system is depicted in figure 2.1. In such a system, defined in Fuggetta et al. [34] as a **True Distributed System (TDS)**, components are mostly unaware of their physical location in the network, or they even do not know anything about the underlying network. When component A invokes a service on component B, there is no clue about which node B is running on and the interaction is perceived as a local one. Systems based on CORBA [52] are well-known examples of TDSs, where the physical location of components is intentionally hidden to the programmer of the distributed application.

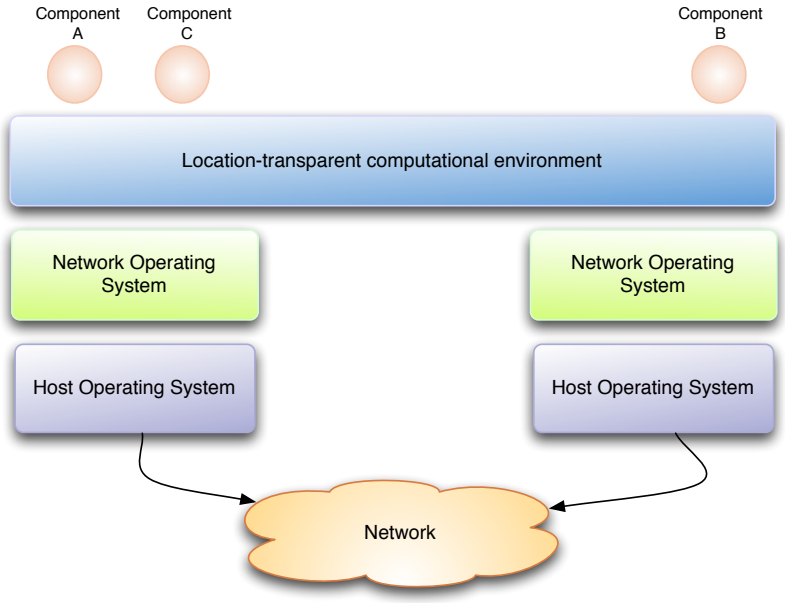


Figure 2.1: A high-level view of a True Distributed System hiding location to deployed components

Nonetheless, the technological and methodological background developed for such conventional distributed systems often fails to scale up when applied to large scale distribution. In these scenarios, the concepts of *location* and *mobility* have been deemed so important that in mobile code paradigms they affect the conceptual structure of the application as it is

conceived in the design stage. From the perspective of a **Mobile Code System (MCS)**, components should be location-aware and possibly take actions based on such knowledge (see figure 2.2). In large scale distributed systems, providing location transparency to software components may lead to unexpected performance and reliability problems, mainly because of the nature of interactions involved.

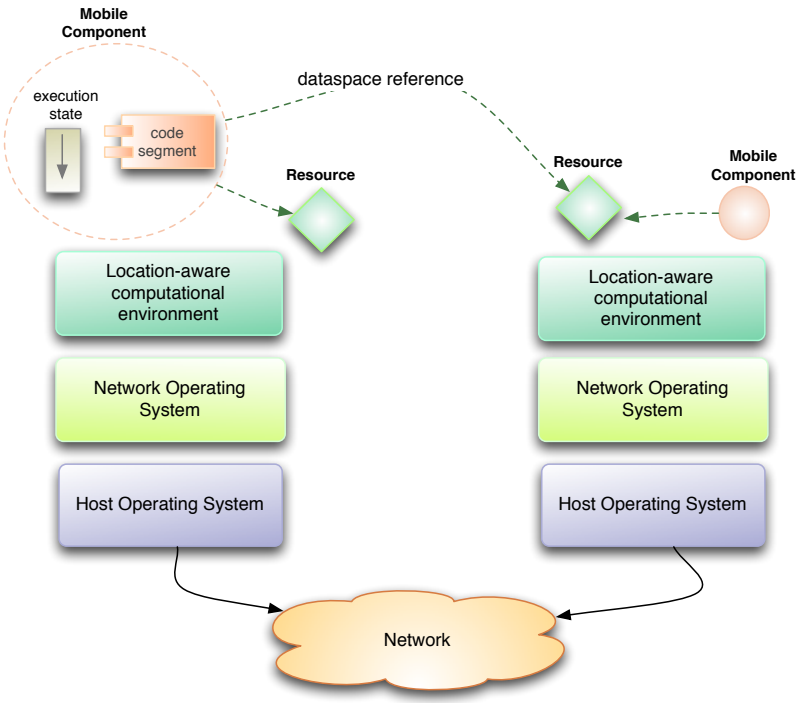


Figure 2.2: A high-level view of a Mobile Code System providing components with location awareness and mobility

Interactions among software components on the same machine are remarkably different from interactions among distributed components, in terms of latency, access to memory, partial failure and concurrency. A number of researchers both in academia and industry [39, 34, 46] have acknowledged that *mobile components* (i.e. software components able to move from one node to another on need) in a MCS can significantly overcome the fragility and lack of flexibility of traditional client-server interactions when occurring in a large-scale distributed system. Some well-assesed benefits are for instance:

- *Enhanced flexibility.* Clients typically access the resources hosted by

a server through a predefined set of services, whose interface is determined a priori and agreed among the client and the server. It is usually complex, if at all possible, to change dynamically this interface to encompass new services, or to adapt the existing ones to previously unforeseen needs. Code mobility can be used to update dynamically the interface on the client and/or server side.

- *Reduced bandwidth consumption.* The ability to migrate a client program in a MCS to achieve co-location with the resources it must access on the server reduces the need for remote communication and this may enable, under some conditions, a more efficient use of the communication link.

- *Improved fault-tolerance.* In conventional systems, a high-level interaction between a client and a server unfolds as a series of pairwise low-level interactions under the form of requests and replies. During these actions, the state of the overall computation is distributed. This fact heavily complicates the task of recovering from a fault. Instead, in a MCS, mobile components embed the code describing the whole high-level interaction and they can migrate on the server. Thus, the state of the interactions remains entirely local, and faults can be dealt with easily, e.g. using checkpointing.

- *Support for disconnected operations.* The capability to act autonomously from the application that dispatched them, makes mobile components very useful in scenarios characterized by physical mobility and very frequent network disconnections.

- *Protocol encapsulation.* In conventional systems, data is typically a passive element that gets processed by other active components in the system. Code mobility has changed this view dramatically, by allowing a piece of data to travel within the system together with the application logic needed to interpret and manipulate it. Clearly, this possibility greatly improves the flexibility of the system, by simplifying the deployment of different, co-existing policies for using data.

Summing up, location-awareness can therefore be considered a precious strategical information and mobility the main tool to tackle the intrinsic unreliability of large-scale distributed systems and to boost flexibility in software applications.

2.1.1 An abstract model for mobile components

In order to perceive the difference among mobile code paradigms, some key abstractions are essential. A *Computational Environment (CE)* is the middleware supporting the execution of mobile components. It is present in

each node of the network and embodies the notion of location mentioned before. The other two abstract elements of the model are *resources*, representing passive data (like database files or sockets) or physical devices, and *execution units (EU)*, as the abstraction of a computational flow. Resources are accessed by EUs and possibly shared by them. An EU is conventionally split into three separate parts that are supposed to be movable to achieve the overall mobility of the execution unit (see figure 2.2):

- the **code segment**, i.e. the instructions to be executed. In an object-oriented system, it is the set of compiled methods of the application;
- the **dataspace**, a collection of all the resources accessed by the execution unit. In an object-oriented system, these resources are represented by objects in the heap;
- the **execution state**, containing private data as well as control information, such as the call stack and the instruction pointer.

In convential systems, each EU is bound to a unique CE for its whole lifetime. Moreover, the binding between the EU and its code segment is generally static. Even in the presence of dynamic linking languages, the linked code is a resource of the current CE. This is not true for a MCS. MCSs are characterized by the fact that the code segment, execution state and data space of an EU are able to migrate from CE to CE. A key distinction for this dissertation thesis is between **strong mobility** and **weak mobility** of EUs.

- **Strong mobility** is the ability of a MCS to allow EUs to move their code and execution state to a different CE. Execution units are suspended, transmitted to the destination CE and resumed there.
- **Weak mobility** means that the MCS allows the EU to be bound dynamically to code coming from a different CE. The code can be accompanied by some initizialization data, but no migration of execution state is involved.

Furthermore, EU migration can be either **proactive** or **reactive**. A proactive migration is requested by the migrating EU itself, which can decide the destination node as well. In reactive migration, the movement of the EU is triggered by another EU which can act like a master EU, e.g. in a load balancing system.

2.1.2 Resource relocation of an execution unit

In both strong and weak mobility MCSs, when the state of an EU is moved, what happens to referenced resources in the source CE? In other words, what happens to the whole dataspace of the source EU upon migration?

Two classes of strategies have been classified [34]: **sharing strategies** and **replication strategies**. A **sharing strategy** implies that the original binding is kept and therefore inter-CE references (known as *network references*) to remote resources must be generated. **Replication strategies** can be further divided in:

- **Static replication strategies**. Some "ubiquitous" resources can be statically replicated in all CEs. System variables and user interface libraries are good examples of such resources. The original bindings to such resources are deleted and new default bindings are established with local instances on the destination CE.
- **Dynamic replication strategies** A copy of the bound resources is made in the destination CE, the original bindings are deleted, and new bindings are established with the copied resources. Two further options exist: (i) removing the bound resources from the source CE (*replication by move*) or (ii) keeping them (*replication by copy*).

MCSs may exploit different strategies for different resources. Sharing is adopted for resources that have to shared among EUs, on different CEs. The sharing strategy leads to state distribution. In fact, when this strategy is adopted, the data space of the remote EU contains resources located in the source CE. Static replication can be used only for stateless resources or for resources whose state has not to be maintained consistent across CEs. Dynamic replication by copy is adopted to ensure resource availability both on the source and destination CE. Dynamic replication by move is chosen for resources that are neither to be shared nor to be available on both the origin and destination CE. Otherwise, when a resource vanishes, a dangling reference can arise. In addition, dynamic replication is adopted for simple values like integers or strings.

2.2 Mobile code paradigms

Mobile components are expected to provide services to end-users or to other components in the distributed software application. In the previous section, the three fundamental elements of a mobile component have been discussed. These elements are the know-how about the service (i.e. its *code segment*), the resources that are needed to carry it out (i.e. the *dataspace*), and the "active part" in charge of realizing the service itself (i.e. the *execution state*). Clearly, a service can be effectively performed only when these three capabilities are co-located. In a pure client-server paradigm, these capabilities are permanently co-located by design on the server node, and are exploited by the client. Instead, mobile code paradigms identified in [27] provide different strategies for relocating these capabilities at runtime.
In the **Code On Demand (COD)** paradigm, one of the two components

(e.g. the client) lacks the know-how about how to perform the service, although it owns the necessary resources. The corresponding code is then downloaded from a remote server acting as a code repository, and subsequently executed. This paradigm provides enhanced flexibility by allowing the server to change dynamically the behaviour of the client (or vice versa). For instance, this is the scheme typically employed by Web applets.
In the **Remote Evaluation (REV)** paradigm, the client owns the know-how about the service, but lacks the resources necessary to its execution, which are owned by the server component. As pointed out in [65] a sort of enhanced client-server interaction takes place, where the client includes in the request to the server also the code required to perform the service. After this code is received and its execution has started on the server, the interaction proceeds as in the client-server paradigm: the code received is able to access the resources now co-located with it, and eventually send the results back to the client.
In the **Mobile Agent (MA)** paradigm the client knows how to perform the service but lacks part of the resources, which are owned by the server. The client then autonomously migrates to become co-located with the server, and thus to perform the service by exploiting local access to resources.

2.3 Mobile Agents and the Java language

MAs are the most innovative and challenging of the three paradigms discussed so far. The very idea of an "autonomous, proactive and intelligent entity" (i.e. a *software agent* [42]) has always been a fascinating one for many researchers of different areas all around the world. Agent-based research is exploring (and realizing) the idea of software components acting like individuals in human societies, with rational and social capabilities. The promise of agent-based software engineering is to offer a new way of designing software applications by means of an abstraction (i.e. the agent) closer to the human behaviour than to the physical machine.
However, the focus of the present work is more on the *mobility aspect* of agents and its implementation in the Java language [23]. A significant number of languages have been created or simply adapted to implement the above mobile code paradigms. Surveying those languages in this work would be out of scope, yet the interested reader can refer to [30] for an exhaustive and authoritative treatment of the topic. Anyway, the key role played by the Java language in the last decade is beyond dispute.

Mobile agents are basically implemented through Java threads [51] directly or indirectly. Threads are considered a valid example of EU, performing their tasks in a computational environment (i.e. the Java Virtual Machine), but without any possibilities of detaching from their native envi-

ronment. The **Java Virtual Machine (JVM)** [47] has always been chosen as the best CE for mobile agent platforms because it offers an extremely effective set of "functional" and "non-functional" features.
The use of a portable intermediate language (known as *bytecode*) and the support for object persistence (called *serialization*) are the "functional" aspects that made Java's fortune. On the one hand, the code segment of a Java agent (made up of the bytecode of all invoked methods) is portable by definition among different hardware architectures and operating systems. URL classloaders allow agents to dynamically fetch their code from a remote host and execute it locally. On the other hand, the object serialization protocol provides an easy and platform-independent way to move Java objects from one heap to another. Assuming that all the objects referenced by an agent are "serializable", its dataspace can be relocated and then moved from one node to another one with little effort.
Among the set of "non functional" features of the Java language and its JVM, it is worth mentioning the *sandbox security model* and the *bytecode verification* [71]. Thanks to the sandbox model, it is possible to limit the set of local resources that an agent can access and the privileges on such resources. Moreover, bytecode verification allows to intercept potentially malicious or malformed code before it is executed by the interpreter of the JVM. Research on these features [20] is of paramount importance to the general acceptance of the MA paradigm, especially outside the academic community. The skepticism towards MAs [80] in industry is mostly related to the lack of powerful security, trust and reputation mechanisms in current mobile agent platforms.

2.4 Mobility in Java: strong or weak?

To this point, it should be clear that migrating the code segment and the dataspace of a Java thread is mostly a solved problem and for this reason there is little research to do on this topic. How to easily and efficiently relocate the execution state of a thread is instead still a debated topic in the mobile code community. What is generally called the execution state in the case of a Java thread includes the call stack, the program counter and a bunch of machine registers. According to the architecture of the JVM [47], threads are objects with a deep and strong binding to the underlying operating system. For efficiency and security reasons, user programs can access only a limited and safe set of methods to manage Java threads. Structures like the call stack and machine registers are visible only to the JVM runtime and cannot be extracted or modified from the outside. Java threads are in fact coherently presented to the programmer as objects, but their serialization does not produce the desired effect of "capturing their current execution flow and resuming it elsewhere". Among the three EU parts (i.e.

code, dataspace and execution state), the execution state is the most difficult one to move, because Java does not provide any built-in mechanism to support it.
As already stressed in subsection 2.1.1, there are conceptually two approaches when dealing with the migration of the execution state. As concerns Java, **strong mobility** aims at finding a mechanism to capture the execution state from a running Java thread and restore it on another JVM. As it will be described in the next chapter, some researchers chose to attack the problem from outside the JVM (e.g. instrumenting the bytecode of the mobile agent), while others chose to deal with it from the inside (e.g. modifying the bytecode intepreter to keep track of the execution state).
Nevertheless, it has been acknowledged [25] that for some typologies of applications the weak form of mobility is enough. **Weak mobility** can be in fact viewed as a "workaround" to the complexity of a strong state capture. Weakly mobile threads can transfer their execution, bringing only code and some data, while the call stack is lost.
From the architectural standpoint, it is quite easy to implement weak mo-

```
public class MyAgent extends Agent {

      // indicates if the agent has moved yet
      protected boolean migrated = false;

      public void run(){
          if( ! migrated ){
              // things to do before the migration
              migrated = true;
              try{
                  migrate(newURL(‘‘nexthost.unimore.it’’);
              }catch(Exception e){
                  migrated = false;
              }
          }
          else{ // things to do on the destination host
                // possibly other if/else to handle
                // other conditions
          }
      }
}
```

Figure 2.3: An example of a weak mobile agent in Java

bility on top of the JVM, thanks to the aforementioned object serialization

and URL classloaders. From an application point of view, weakly mobile systems usually force the programmer to write code in a less natural style: extra programming effort is required in order to manually save the execution state, with flags and other artificial expedients. Nonetheless, the bulk of Mobile Agents Systems (e.g. Jade [59], Aglets[1]) support only weakly mobile agents, mainly because it is considered enough in many scenarios: e.g. distributed information retrieval, online auctions and other systems where agents have to follow the user's movements, for migrating from and to the user's portable device (mobile phone, PDA, etc.).
A simple weak agent is shown in figure 2.3. The point is that, with weak mobility, it is as the code routinely performs rollbacks: after a successful `migrate()` invocation, the agent (on the arrival node) does not continue its execution in the `run()` method from that point. Instead, the code restarts from the beginning of the `run()` method (on the destination machine, of course), leading basically to a code rollback. The fact that an agent restarts its execution always from a defined entry point, could produce awkward solutions, forcing the developer to use flags and other indicators to take care of the host the agent is currently running on.
A strongly mobile thread has instead the ability to migrate its code and execution state, including the program counter, saved processor registers, return addresses and local variables. The active component is suspended, marshaled, transmitted, unmarshaled and then restarted at the destination node without loss of data or execution state. The choice of strong mobility, when designing distributed Java applications, has to be carefully motivated, since it is not always the best one in most simple cases: e.g. many mobile agent applications [23] do not require such a big support for computations migration, relying on simpler form of data migration. The category of distributed and parallel computations can be considered perhaps the "killer application" for thread mobility. For instance, complex computations, possibly with a high degree of parallelism, carried out on a cluster of servers would certainly benefit from a thread migration facility in the JVM. Wellknow cases of such applications are mathematical computations, which are often recursive by their own nature (e.g. fractal calculations) and can be parallelized to achieve better elaboration times.
In the cluster depicted in figure 2.4, several mobile threads are spawned by a supervisor thread on a master server and each one can be assigned a portion of a huge data space (e.g. temperature and pressure values from different geographic areas, in a weather forecast application). In order to cover the entire data domain, each mobile thread can exploit the migration support by the underlying JVM to move spontaneously (i.e. proactively) from one server (initially the master) to another slave server, without having to restart from the beginning (i.e. strong mobility). It simply carries its current call stack with itself and continues execution at destination from the last executed instruction. In a similar scenario, we can also have mobile

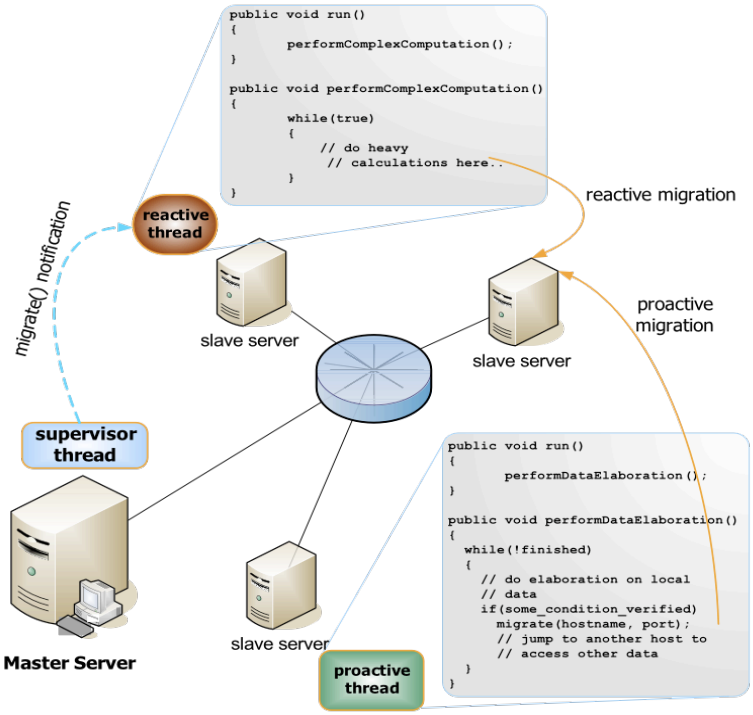


Figure 2.4: A cluster of servers to perform distributed computations

threads moved reactively, i.e. after being notified a migration request from some other thread. This is the case of distributed systems (e.g. in the Grid Computing field [16]) where load balancing is intensively carried out and reactive thread relocation is a must: in such cases a number of slave nodes (like those in figure 2.4) have several tasks assigned to them. In order to avoid overloading some nodes while leaving some others idle (for a better exploitation of the available resources and an increased throughput), these systems need to constantly monitor the execution of their tasks and possibly re-assign them, according to an established load balancing algorithm. In these latter systems, it frequently happens that a supervisor thread manages the workload of each slave server, implementing some load balancing algorithm. The supervisor thread in figure 2.4 should be able to notify one spawned mobile thread to move on a less overloaded server: this clearly stands for preempting its execution at some point in the code, moving its captured state at the destination host and transparently resuming it.
Eventually, it appears evident how strong mobility can be far more powerful than weak mobility, as it preserves the traditional programming style of threads, without requiring any code rollback or other expedients. Despite these advantages, many systems support only weak mobility and the reason lies mainly in the complexity issues of strong mobility and in the insufficient support of existing JVMs to deal with the execution state.

Chapter 3

Implementing strong mobility in Java: Approaches and techniques

After having introduced mobile code and strong mobility, this chapter goes deeper into the mechanisms and the strategies experimented to enable strong thread migration in the Java language. The first section gives therefore some hints on the *Java Virtual Machine (JVM)*'s architecture, which is the essential precondition to grasp the differences between the different approaches in the literature.

3.1 The JVM runtime at a glance

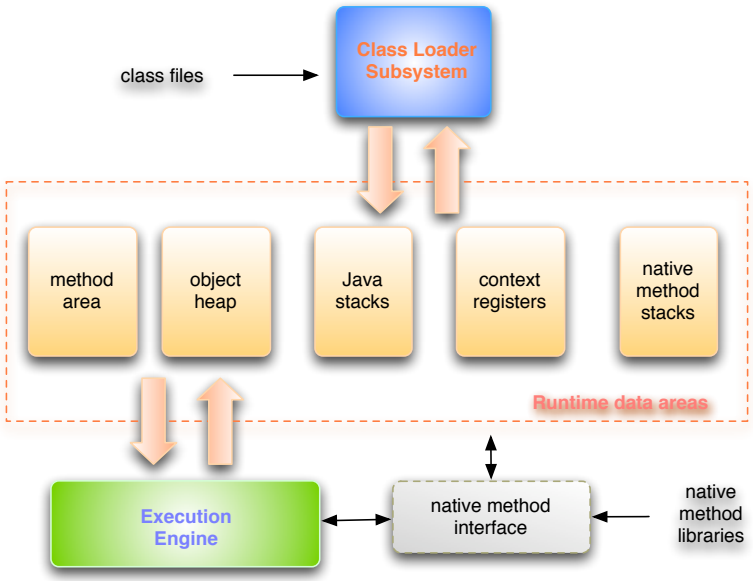


Figure 3.1: A conceptual architecture of the JVM

The JVM [47] is an abstraction of a homogeneous machine with an *execution engine* (the equivalent of a hardware processor), intepreting a defined set of instructions (*bytecode*), and with *runtime data areas*, e.g. used for memory and process management (see figure 3.1).
The Java *bytecode* provides an instruction set that is very similar to the one of a hardware processor. Each instruction specifies the operation to be performed, the number of operands and the types of the operands manipulated by the instruction. For example, the `iadd`, `ladd`, `fadd` and `dadd` instructions respectively apply on two operands of type `int`, `long`, `float` and `double` and return a result of the same type. The execution of the bytecode in the JVM is based on a stack, called the *operand stack* internal to the method frame. If, for instance, an `iadd` instruction is executed by the execution engine, two integers are pushed on the operand stack, and after the operation

is completed the integer result is left on top of the stack.
As for the *execution engine* (see figure 3.1), it must pointed out that in its first releases the JVM was based on an interpreted scheme, in which the JVM interpreter traslated each bytecode instruction in the execution of native code (just like a sort of scripting language, e.g. bash scripts). Second-generation JVMs (e.g. Sun's Hotspot) integrated the so called **Just-In-Time compiler (JIT)**, which compiles on demand Java methods into native code, so as to execute them at full speed. This innovation greatly improved the performance of the execution engine and it represents nowadays the standard for every JVM implementation.

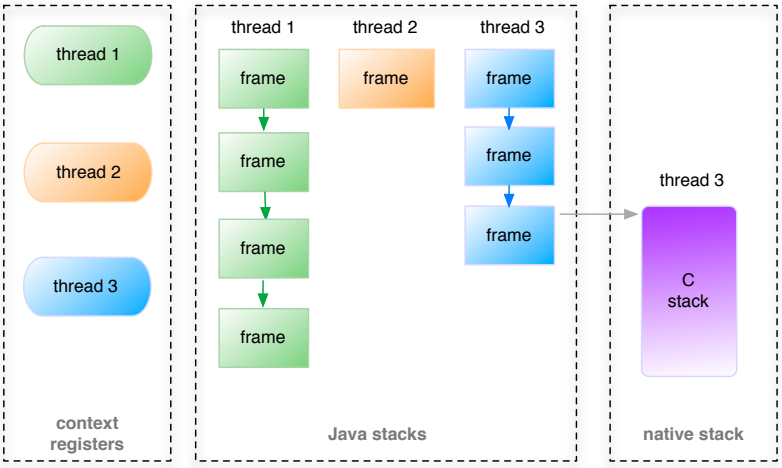


Figure 3.2: The Java method stack and its frames

Lindoholm and Yellin [47] precisely defined the *runtime data areas* of the JVM as composed by five main elements (figure 3.1):

- *Context registers*, storing the values of machine registers (e.g. the return address) as they were saved when the thread was preempted by the JVM scheduler.

- *The Java stacks* (in figure 3.2). Each thread in the JVM has its own private Java stack, made up of *frames*. A new frame is pushed on the stack each time a Java method is invoked by the thread and popped from the stack when the method returns. A frame includes a table containing the local variables of the associated method and an operand stack with the partial results of the method (e.g. the two operands of an `iadd`). The values of local variables and operands may be of several types, like integers, float or object references. A frame contains also

context registers such as the program counter (i.e. the return address in the caller method) and the caller frame pointer.

- *The object heap.* It is an area in the address space of the JVM OS process used to perform dynamic allocations of objects at runtime. Objects no more referenced by any thread in the system are collected by a special component of the JVM, called Garbage Collector [82].
- *The method area.* This area includes all the classes loaded by the JVM and their methods, compiled by the JIT compiler into native code of the underlying architecture.
- *The native stacks.* When a thread invokes a `native` method (e.g. one from a C/C++ library), the execution state of the thread is no more confined to its Java stack but it is handled by a special native method stack. The JVM specifications [47] does not provide any details about the implementation of the native stack, whose structure is by definition platform-dependent and left to the JVM designer.

3.2 The phases of thread migration

The previous section has briefly sketched the architecture of the JVM, focusing on those components that are directly involved in the migration process. This section traverses the main phases involved in a thread migration, no matter the approach or the techniques used. Schematically, *thread migration* means basically

1. freezing the execution of a chosen Java thread,
2. safely and completely capturing its execution state,
3. trasferring such a state using some kind of protocol,
4. recreating a new thread initialized with that state.

Thread migration is expected to act as fast as possible in order to be truly effective in its typical application scenarios (e.g. load balancing or checkpointing). All the enumerated phases contribute to some extent to the final performance and agility of the migration process. In the following subsections, a concise description of each phase is given, so as to pave the way for the survey of related work in section 3.3.

3.2.1 Interrupting a running thread: proactive and reactive migration

As already introduced in section 2.1.1, a thread migration can be started willingly by the thread (i.e. *proactive migration*) or it can be triggered

by another thread (i.e. *reactive migration*). An application scenario, in which reactive migration plays a major role, is a load balancing facility in a distributed system. If the virtual machine provides such a functionality to authorized threads, a load monitor thread may want to suspend the execution of a worker thread A, assign it to the least overloaded machine and resume its execution from the next instruction in A's code. This form of transparent externally-requested migration is harder to implement with respect to the proactive case, mainly because of its asynchronous nature. Proactive migration raises, in fact, less semantic issues than the reactive one, though identical to the latter from the technological/implementation point of view. The fundamental difference is that proactive migration is synchronized by its own nature: the thread usually invokes some sort of `migrate()` method when it wants to migrate. Yet in a reactive migration the time when the thread has to be interrupted could be unpredictable (the requester thread notifies the migration request to the destination thread, but the operation is not supposed to be instantaneous).
Therefore, in reactive migration, the critical design-level decision is about the **granularity of the interruption** to provide. In a few words, the question is: should the designated thread be interruptible anywhere in its code or just under certain safe circumstances? If we allow a migration with too fine a granularity, inconsistency problems are likely to occur: the thread can potentially lose control in any method, from its own user-implemented methods to internal Java library methods (e.g. `System.out.println()`, `Object.wait()` and so forth). It may also occur that a critical I/O operation is being carried out and a blind thread migration would result in possible inconsistency errors. To this purpose, a clear semantics of interruptability has to be devised: a simple and popular approach is to introduce special keywords in the source code language to mark methods and classes that can be interrupted. Yet, keywords and ad hoc constructs seriously hamper the portability and usability of the framework itself.

3.2.2 Extracting the state of a thread

Capturing the execution state of a Java thread boils down to reconstructing its program counter and the sequence of Java frames in the call stack. Extracting those information is somehow the reverse operation normally performed by the execution engine: instead of converting the bytecode-based execution flow of the thread into a machine dependent one, stack capturing means rebuilding a bytecode-level representation of a native execution flow. The main issue to cope with in state capturing is the **observability of thread state**. In a few words, it should be possible to map the physical execution state at a certain point into a sequence of Java frames. The physical implementation of a Java frame is left to the JVM designer and may not always be observable during the execution of the thread.

In *interpreter-based JVM*s (as explained in section 3.1), the runtime execution engine has to repeatedly decode the bytecode, fetch the operands and then execute the instructions until the program exits. Each bytecode instruction translates to several native instructions (i.e. machine assembly code), depending on the underlying computer architecture (e.g. IA32, PPC or SPARC). The execution state during such native instructions is not observable at the bytecode level and the only choice is to delay migration until the interpreter finished executing the current bytecode instruction and is ready to fetch the next one. This kind of points can be defined as **observability points**.
When *JIT-compilation* is at work, this problem is even tougher, because the entire bytecode of the method is translated into assembly code at compilation time. The outcome is that the thread is no more executing step-by-step bytecode instructions, but just native instructions. Observability points in this case cannot be univocally designated, because JIT compilers usually perform code optimizations (e.g. method inlining) that disrupt the mapping between bytecode and assembly instructions. The key implementation decision is therefore to carefully identify a reasonable number of observability points into method bodies, according to some empirical criteria (e.g. in method prologues or epilogues) and taking care of not hampering the performance of the produced code.

3.2.3 The transmission of the extracted state

Once the state has been captured in a sequence of Java frames, the migration framework has to properly handle its transmission to another JVM. To this purpose, most Java-based migration frameworks rely upon the standard Java serialization protocol, rather than inventing a custom protocol. Java serialization is easy to use and supported by every JVM; besides, the serialized form of objects and primitive types (e.g. integers or float) is completely platform independent. Whenever a Java frame is serialized into a stream, every object referenced by its local variables and stack operands is serialized in its turn. Thanks to its recursive nature, Java serialization makes it straighforward to transfer the dataspace of the thread, along with its execution state.
A possible drawback of Java serialization is the relatively **poor performance of the protocol**, mainly due to its verbosity. Each and every data written is accompanied by descriptive metadata used to correctly interpret the stream at deserialization time. Some research work on this topic [19] has been done to tackle the problem in specific contexts, like high-performance computing and distributed systems. In [29] the key to booster the serialization process is usually in the adoption of a direct memory to memory dump in replacement of the usual marshalling of structured values. Researchers in [8] chose instead to trade off portability for faster times, implementing a

serialization protocol tailored to a specific virtual machine (i.e. JikesRVM).

3.2.4 Restoring a migrated thread in the JVM

The last phase of the migration process is the restoration of the extracted state on the destination JVM. Assuming that the bytecode of the thread has already been cached locally, the restoration of the execution state automatically restores every serialized object into the destination heap. The only thing that has to be manually installed is the execution state, i.e. the call stack and the context registers of the thread being migrated.
One way to rebuild the call stack is to manually fill in an empty stack with all the frames and then create a new thread running on that new stack. Nonetheless, this approach is a very fragile one, because it strongly depends on a specific stack layout. A very common solution is using special **method stubs** that are executed by the new thread. Their purpose is to re-establish the original call stack up to the next instruction to be executed. It is like running again the past execution of the thread with the fast-forward button.

3.3 Related Work in strong Java thread migration

After the explanation of all the steps and issues involved in a thread migration, in this section a critical review of related work in this research field will be proposed. The discussion starts with an historical survey of the main Java-based systems in literature, moving then to a more critical comparison based on some identified criteria.

3.3.1 The historical outline

In the research community, implementations of process/thread migration in programming languages appeared well before Java gained its popularity. Since 1998, the idea was already being experimented using languages like Emerald ([43, 68]) or later in 1997 with Agent TCL [44] (a scripting language based on the TCL language [53]) and Ara [55].
For our scope, however, we are interested in Java-based solutions. It was in fact in 1996 that the first Java-based implementation of strong mobility was released with **Sumatra** [10]. Sumatra was an extension of Java that supported resource-aware mobile programs. The aim of Sumatra was to leave the newly born Java language untouched, providing all the functionality needed to implement thread migration through an extension of the Java class library and modifying the Java interpreter. Sumatra was implemented as a research project at the University of Arizona [9] and is no more active.

After the release of Sumatra, it was clear that strong mobility was still too an immature concept to be adopted as a standard JVM extension. Suma-

tra was an interesting Java interpreter, but it posed heavy trust and security problems. Many researchers tried therefore to add strong thread migration to existing JVMs without touching the JVM implementation, but simply manipulating the application code through some kind of *code preprocessor*.
Wasp [35] was released in 1998 by the Darmstadt University of Technology (Germany). It is equipped with a special preprocessor that instruments the user code by adding hidden Java statements that capture the actual state and reestablish the state on restart at the target machine. The original program code was parsed with the JavaCC-tool [6] from a Java 1.1 grammar. **JavaGo** [62], developed at the University of Tokio, used a similar source-to-source translation to achieve thread migration. JavaGo extended the Java language by adding three constructs (based on mobile calculus [63]) for controlling migration. The most interesting one is the `migratory` method modifier, added by the programmer to methods that are to be transformed for migration. This modifier makes source-code translation easier, with respect to Wasp that lacks such kind of declaration. **Chakravarti** [28] and **STRING** [48] use source code transformations to translate application code using strong mobility constructs into weak mobility code: they save the current execution state into the source code and send it as a set of serialized objects, like in weak mobile agent platforms.

From source-to-source instrumentations, in 2000 the interest focused more on bytecode-level transformations. Code produced by bytecode transformers shows better execution efficiency than code produced by a source code transformer. The very JavaGo system was in fact extended by its authors to use a bytecode preprocessor, producing **JavaGoX** [61]. The bytecode preprocessor takes a method in bytecode and produces another one that has instructions for saving and restoring its execution state. **Brakes** [79], developed at the Katholieke Universiteit of Leuven (Belgium), shares the same technique as JavaGoX.

Another branch of researchers in strong mobility followed the path traced by Sumatra and from 2000 to 2002 further explored state capturing and restoration from inside the JVM. It happened, for instance, with the **Nomads** [74] mobile agent system, developed at IHMC [5] (Institute for Human and Machine Cognition) in Florida. Nomads runs on top of the **Aroma VM** [72], a Java-compatible VM that provides unique capabilities such as thread and VM state capture and dynamic, fine-grained resource control and accounting. Aroma was designed with the specific purpose of allowing multiple concurrent threads to be checkpointed (execution state of a process saved to some form of storage).
Merpati [69], **D'Agents** [38] and **ITS** [17] are other examples of a modified JVM interpreter. Merpati was developed at the University of Zurich and is based on the Sun JDK 1.1.5. Merpati provides a thin API that comprises

mechanisms for checkpointing, recovery and migration of a JVM. D'Agents is instead developed by the Darthmouth College (US). It supports multiple languages, Tcl, Java and Scheme, and strong mobility for Tcl and Java. ITS was developed at INRIA (France) and has been integrated in the JVM by extending the JDK 1.1.3.
CIA [41] (from the University of Ulm, Germany) is instead based on a modification of the JVM Debugger Interface (JVMDI) [70] to perform state capturing using debugging information produced by the `javac` compiler in the Sun JDK and maintained by the debugger of the JVM.
After JIT compilers became a breakthrough in the JVM design, thread migration tried to evolve and keep up with the new technology. As already mentioned, capturing the state of a thread in presence of JIT compilation is quite a challenging task. This is perhaps the reason why only two full-fledged JIT-based migration systems have been proposed so far (not counting this dissertation work). **JESSICA2** [83] and **JavaThread** [18] have both been released in 2002. JESSICA2 (from the University of Hong Kong) implements thread migration using dynamic native code instrumentation and dynamic register patching, and is mainly used as a distributed JVM to enable automatic thread distribution and load balancing in a cluster environment. JavaThread is instead a JIT-based evolution of ITS. It uses a type inference technique, without instrumenting the bytecode or interferring with the JVM interpreter, and relies upon dynamic de-optimization [50] to transform physical frames associated with JIT compiled methods into Java frames.

3.3.2 Comparing approaches and techniques

The research work presented in the previous subsection is now analyzed according to the technique used to externalize the execution state of the migrating thread. Browsing the systems above, two big categories can be identified (see figure 3.3):

1. the **JVM-level approach**
2. the **application-level approach**

The JVM-level approach means basically modifying or extending the source code of existing JVMs to introduce APIs for enabling migration. The application-level approach is instead based on some kind of source code transformation to constantly trace the state of each thread and to use the gathered information to rebuild the state remotely.

JVM-level approach

The former approach is, with no doubt, more intuitive because it provides the user with an advanced version of the JVM, which can completely exter-

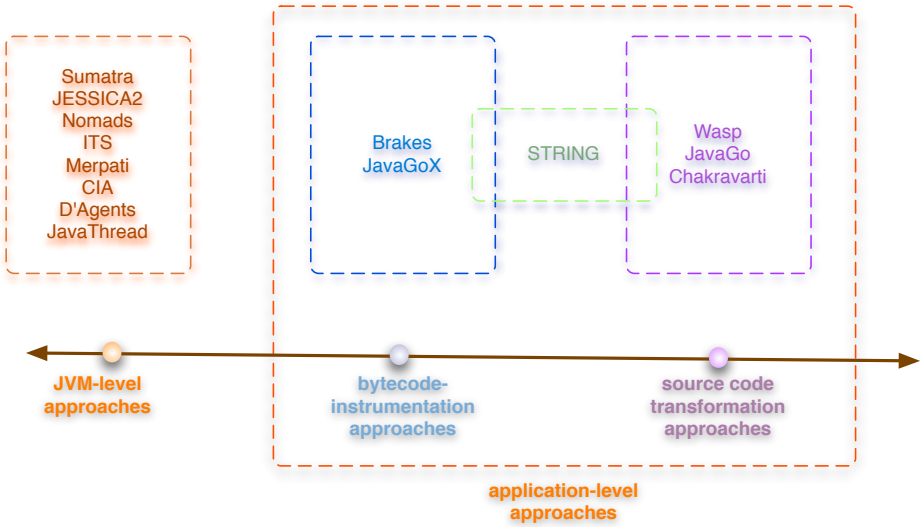


Figure 3.3: Classification of strong mobility approaches in Java

nalize the state of Java threads (for thread serialization) and can, furthermore, initialize a thread with a particular state (for thread de-serialization). The modified JVM should, first of all, define **observability points** with their granularity and then a mechanism to track the state at those points. Such points (introduced in section 3.2.2) are places in the executed code where the mapping between the physical stack and the bytecode-level variables (locals and stack operands) is visible. They can be built either at **runtime time** (i.e. during the execution of the application code) or at **migration-time**.

The **run-time strategy** is very frequent among interpreter-based JVMs, because the execution engine sequentially executes bytecode instructions and can thus easily keep track of what local variables are alive or what elements are on the operands stack at the current point. To automatically marshal the stack, the interpreter in **Sumatra**, **D'Agents** and **ITS** maintain a type stack parallel to the physical stack and, when the thread migrates, it transports with the thread primitive values and objects referenced by the stack. It must be pointed out that actualizing type information at run-time strongly impairs the performance of the JVM. The more the instructions executed, the higher the loss of performance, since additional computation has to be performed in parallel with bytecode interpretation.

On the other side, **Merpati** is one of the systems that follows the

migration-time strategy. In order to compute the above mapping for every frame in the physical stack, Merpati exploits type information contained into bytecode instructions. The sequence of executed instructions until the migration request is computed by Merpati using a control flow analysis [11]. Then it determins a path between the method entry point and the program counter. Eventually, the JVM simulates the sequence of instructions. The control flow analysis and the simulation are performed for every Java frame on every Java stack. In **Nomads**, when an individual thread state capture is requested, the Aroma VM examines each of the stack frames in the method stack of the thread and saves all relevant values (such as the program counter, the local variables, and the operand stack) as well as all reachable objects. Restoring the state is possible only into a "blank" virtual machine (i.e., a newly created virtual machine without any threads). Migration is transparent and can conceivably occurs between any two instructions. **JavaThread** has a type inference mechanism that works at serialization time. It uses *type frames* to gather type information for every method frame in the stack and *dynamic de-optimization* to map the physical stack to logical Java frames. The type stack is built parsing, for each frame on the thread's Java stack, the bytecode of the associated method and following the entire code path up to the current program counter. Dynamic de-optimization is a technique introduced by the Hotspot VM to debug JIT-compiled code, and it is used by JavaThread to reverse an optimized frame into a non optimized one. One drawback of dynamic de-optimization is that, if state capturing fails, every optimization performed so far by the JVM is lost. The same problem occurs at deserialization time, when the entire migrated thread is restarted in a non optimized form. The idea of **JESSICA2** is to use the JIT compiler to instrument native codes and help the transformation from physical thread state to bytecode-level state. These native codes will spill the most recent information of variables in the stack at some points, and when the migration request arrives, the scheduler can perform on-stack scanning and build the sequence of frames. Eventually, among the JVM-level systems that perform state capture at migration-time, **CIA** must also be included for its use of JVMDI [70]. CIA needs huge data structures and a stand-alone process (different from the JVM process) to collect the thread context. The Java application must be compiled by the Sun JDK *javac* (with a special command-line option) in order to include debugging information required by JVMDI.

The main drawback of every JVM-level solution is that they implement special modified JVMs (thus not portable) that users have often to download; therefore they are forced to run their applications on a prototypal and possibly unreliable JVM.

Application-level approach

In order to address the issue of non-portability on multiple Java environments, some of the reviewed projects propose solutions at the application level. In these approaches, the application code is filtered by a pre-processor, prior to execution, and new statements are inserted, with the purpose of managing state capturing and restoration. Such a portability is however achieved at the price of a slowdown, due to the many added statements.
As depicted in figure 3.3, two categories can be identified in the application-level category. Systems like **JavaGo**, **Wasp** and **Chakravarti** rely upon **source code preprocessing**. The transformation carried out by **JavaGo** and **Wasp** are quite similar and can be summarized as follows: the body of each application method is enclosed with a try statement to capture a special exception raised by the occurence of migration. When the method actually captures the exception, the values of all local variables are stored into a special state object and the exception is raised again, until the exception reaches the bottom of the stack. **STRING** and **Chakravarti** hide a weak mobility system behind the appearance of a strong mobility one: they, in fact, re-organize "strongly-mobile" written code into a "weakly-mobile" style, so that weak mobility can be used instead.
Brakes and **JavaGoX** implement instead application code modifications at the **level of bytecode**. Two reasons for doing this are that

- the modified code is less verbose and more performant than in the source code case
- Java source code is often unavailable because bytecode is deployed instead.

JavaGoX basically shares the same approach of **JavaGo** and the only difference between the two is in the language parsed and instrumented (i.e. bytecode in the former case, Java code in the latter). **Brakes** currently consists of two parts: (i) a bytecode transformer (based on version 1.4 of the ByteCode Engineering Library [57]), which instruments Java classfiles so they are able to capture their current internal state at any given time; (ii) a small framework which uses the ability of the 'patched' classes to allow Java threads to pause and resume whenever desirable.
Last, it must be observed that **STRING** has been collocated in figure 3.3 as a midway approach between the two kind of application-level approaches, because it uses a mix of bytecode and source code instrumentation to achieve strong migration. Anyway, also state capture with a bytecode preprocessor requires complex bytecode analysis (e.g. to determine whether a variable in the frame is valid at some point in the bytecode) and the produced transformations must pass the Java bytecode verifier test.

3.3.3 A short insight into this dissertation work

The approach followed in the present dissertation thesis is now briefly introduced, stressing how it relates to the suveryed approaches and how it manages to address many of their problems. An extensive presentation will be the subject of chapter 5. The present research work has been motivated by **the need to conceive and implement a new Java-based thread migration approach tailored to the needs of high performance distributed systems**. By high performance distributed systems, we mean dedicated clusters of machines on which to efficiently run distributed computations, like robot coordination algorithms or mobile services (these two applications have been concretely experimented in a cyberwar context as explained in chapter 7).
Among the objectives that have driven the formulation of the presented migration approach, we can distinguish between *architectural aspects*:

- Fast and complete state capture and restoration
- Minimum overhead on the execution of the compiled code
- Efficient integration in a JIT-compiled JVM, even in presence of aggressive code optimizations

and *programmability aspects*:

- Concise and simple API to migrate and restore threads
- Support for both proactive and reactive migration
- JDK 1.5 or higher compliance

The first design choice to reach the above *architectural* goals has been to select a suitable and powerful JVM on which to implement the approach. JikesRVM (see chapter 4 for more details) is an open source project, created by IBM and now hosted on sourceforge [7]. Such a JDK 1.5 compliant JVM is almost completely written in Java and it is equipped with an extensible JIT compilation framework specifically thought for research and experimentation in compilation techniques. The migration framework described in this thesis (called **Mobile JikesRVM** [24]) has been designed as a plug-in to JikesRVM's source code and is written in Java as well.
In particular, the plug-in has been conceived as very independent from the rest of the JikesRVM runtime. It basically comprises two modified JIT compilers (i.e. a baseline and an optimizing compiler) to produce "migration-ready" versions of the methods in a mobile thread. No code instrumentation is performed on the compiled code, because the implemented approach relies on tiny compilation maps, partly produced by JikesRVM and partly by the two mobile compilers. Thanks to such maps, it is possible to fully capture

the execution state of threads even in presence of challenging optimizations (e.g. method inlining). Yet, the produced native code does not incur any slowdown due to additional computations (e.g. to update a type stack, like in ITS), because the machine code runs at full speed.
Furthermore, these two mobile compilers are stand-alone components; their code has been installed in the main JIT compilation loop and they are used only to compile classes loaded from a special *mobile code* repository (i.e. a directory configured with an environment variable). This design choice has a twofold effect:

1. Confining any possible overhead to the migrable code. Special compilation does not involve code outside the mobile code repository, which is instead handled by regular JikesRVM compilers.

2. Minimizing dependecies on JikesRVM's source code. The "glue-code" between Mobile JikesRVM and JikesRVM has been limited to three classes. The remainder of the code is independent Java code, disjoint from the JVM's sourcecode.

From an architectural point of view, these two points represent a concrete answer to the doubts and issues raised by JVM-level migration systems. Thread state externalization often requires performing heavy modifications to the JVM runtime, thus raising severe inefficiency and unreliability problems. Isolating modifications helps reducing possible side-effects on the performance of the JVM or undetected security bugs. Last, but not least, this improves the maintanance of the plug-in code, counteracting the fast "aging effect" of those systems. Migration systems based on modified JVMs are nowadays outdated or not supported on newer releases of their JVMs. Sometimes their code is no more tested and fixed, while JikesRVM is being actively contributed by researchers all over the world and, thanks to the above choices, Mobile JikesRVM runs smoothly and independently on top of it (starting from release 2.3.2).

From the *programmability* standpoint, Mobile JikesRVM was explicitly designed to reduce the number and complexity of the API the programmer has to learn. Enabling mobility in Mobile JikesRVM requires simply

- downloading the application bytecode in the mobile code repository
- using the standard Java serialization API even on a thread object

The first operation allows the code to be automatically compiled by the special JIT compilers of the framework. The second point means that the programmer must simply call the `writeObject()` method on an opened `java.io.ObjectOutputStream`, to capture the thread. Then, she can restore its state, like any other deserialized Java object, calling `readObject()`

on a `java.io.ObjectInputStream`. No special construct (like the one in **JavaGo**) or possibly undocumented API is needed. In addition, Mobile JikesRVM provides both proactive and reactive migration, with a careful design of the reactive case to reasonably limit the "degree of interruptability" of the code and avoid spoiling critical operations (as seen in subsection 3.2.1).

Chapter 4

The Jikes Research Virtual Machine

Mobile JikesRVM is the name of the thread migration framework described in this thesis. As anticipated in subsection 3.3.3, the presented approach can be clearly classified as a JVM-level one, with an enhanced support for JIT compilation and code optimizations. Instead of hacking the Hotspot VM by Sun Microsystems, an open source research virtual machine (called **JikesRVM**) has been chosen. The aim of this chapter is to give the reader some key elements of the JikesRVM, which are essential to understand the implemented approach to thread migration in Mobile JikesRVM (see chapter 5). After introducing in the next section the project from a general standpoint, the main features of JikesRVM are presented in sections 4.2 and 4.3.

4.1 The Project

The IBM **Jikes Research Virtual Machine (RVM)** project was born in 1997 at the IBM T.J. Watson Laboratories [13] and it has been recently [14] donated by IBM to the open-source community [7]. Two main design goals drove the development of such a successful and very active research project[12]:

- supporting high performance Java servers;
- providing a flexible research platform "where novel VM ideas can be explored, tested and evaluated".

JikesRVM (originally called **Jalapeño**) is almost completely written in Java. Very little of JikesRVM is not written in Java. The VM runs as a user-level process and, as such, it must use the host operating system to access the underlying file system, network and processor resources. This requires that a small portion of JikesRVM be written in C rather than in Java code (see figure 4.1). The amount of C code is about 1000 lines: about half of this code consists of simple "glue" functions that relay calls between Java methods and the C library (*syscall functions* in figure 4.1). The only purpose of this code is to convert parameters and return values between Java format and C format. The other half of the C code consists of a **bootloader** and two signal handlers. The bootloader allocates memory for the virtual machine image, reads the image from disk to memory, and branches to the image startup code. The first signal handler captures hardware traps (generated by null pointer dereferences) and trap instructions (generated for array bounds and divide-by-zero checks), and relays these into the virtual machine (e.g. as a `NullPointerException` like in figure 4.1). The other signal handler passes timer interrupts to the running JikesRVM scheduler.

Implementing JikesRVM in Java has both development and performance advantages. The major development advantages are those that follow from

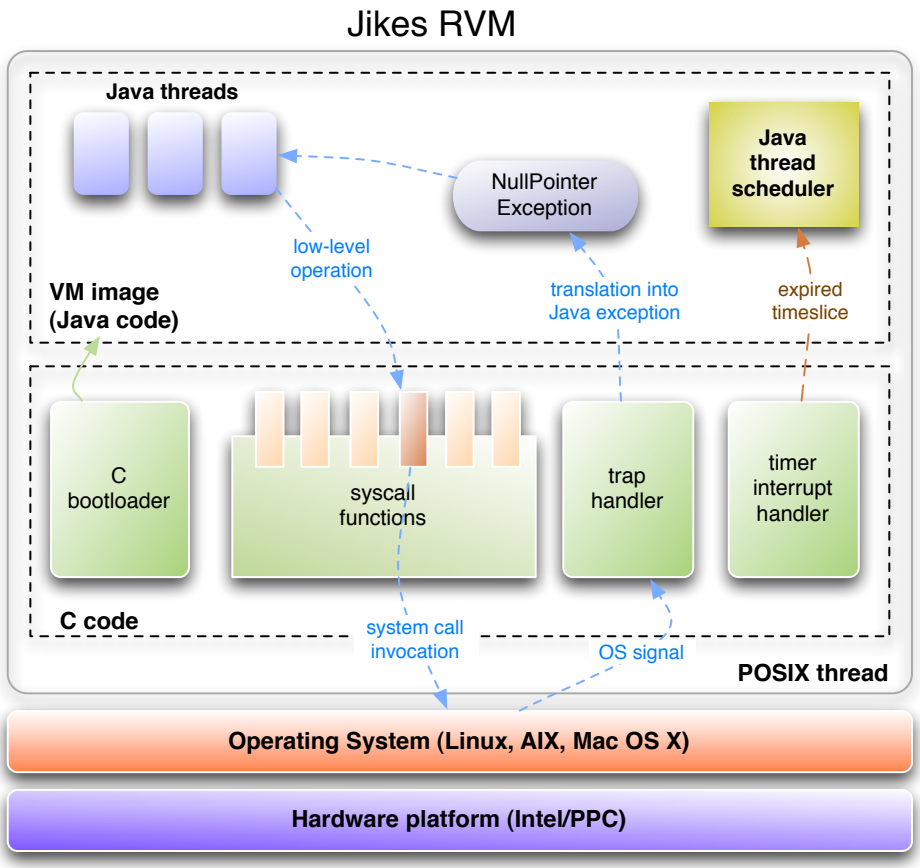


Figure 4.1: The JikesRVM OS process

using a modern, object-oriented, type-safe, and memory-safe programming language. This advantage has been strategic for the development of Mobile JikesRVM in Java. Two kinds of performance advantages have been achieved as well. First, JikesRVM needs no extra code to adapt the call stack between user code and frequently called runtime services: conventional JVMs implement these services using "native" methods (typically written in C, C++ or assembler) incurring an overhead adapting Java state to native state for each call. Second, JikesRVM's seamless operation allows simultaneous dynamic optimization of user code and runtime services, because runtime services are written in Java as well and can be recompiled in an optimized form if necessary.

4.2 The threading model and the "interruptability" issue

Java `Thread`s are programming language abstractions introduced in Java to provide an easy and platform-independent concurrency support to applications. They are created as objects in the JVM heap and launched using the `Thread.start()` method. After calling the latter method, a new execution flow is forked and programs can deal with it through a set of standard primitives, such as `Thread.interrupt(), Thread.join(), Thread.stop()` and so forth.
Even though the behaviour of Java threads is described and standardized in the JVM specifications [47] document, there are no constraints on how to implement that behaviour on a specific hardware and operating system (i.e. the **threading model**). The threading model of the JVM plays a major role in implementing **reactive mobility**, i.e. thread migration triggered by a third thread (see subsection 3.2.1). Choosing to implement reactive mobility means finding a proper and safe way to interrupt the target thread, and enabling a full stack capture at that interruption point.
Two basic threading models have been experimented: the **green-thread model** and the **native-thread model**.

4.2.1 Green threads vs. Native threads

Green threads run at the user level, meaning that the JVM creates and schedules the threads itself. Therefore, the operating system kernel does not know anything about their existence because it sees the JVM only as one thread. Although very lightweight for the operating system, green threads prove inefficient for a number of reasons. Foremost, green threads cannot take advantage of a multiprocessor system. Since the JVM simply runs from within one thread, the threads that the JVM creates are not threads at the OS level. Instead, the JVM splits its timeslice among the various threads

that it spawns internally. Thus, the JVM threads are bound to run within that single JVM thread that runs inside a single processor.
In contrast, **native threads** are created and scheduled by the underlying operating system. Rather than creating and scheduling threads itself, the JVM creates the threads by calling OS-level APIs. As a result, native threads can benefit from multiple processors. Performance can improve because an IO-blocked thread will no longer block the entire JVM, as it happens with the green-thread model. The block will only involve the thread waiting for the IO resource.
In Java 1.1, green threads were the only threading model used by the Hotspot JVM. Since green threads have limitations compared to native threads, subsequent Hotspot versions dropped them in favor of native threads.

4.2.2 The M:N threading model of JikesRVM

In JikesRVM, all Java threads (application threads, garbage collector threads, etc.) are multiplexed onto one or more *virtual processors*, implemented as operating-system threads (i.e. POSIX threads [22]). The number of JikesRVM virtual processors to use is a command line argument and, if no command line argument is given, JikesRVM will default to creating only one virtual processor. Multiple virtual processors require a working pthread library, each virtual processor being bound to a pthread. The JikesRVM is a **M:N threading** model, scheduling execution of an arbitrarily large number (M) of Java threads over a finite number (N) of virtual processors. This means that at most N Java threads can be executing concurrently (true concurrency requires that the underlying platform incorporate multiple execution contexts capable of running several pthreads simultaneously). For maximal performance, the user should tell JikesRVM to create one virtual processor for each CPU on an SMP (Shared-memory MultiProcessor) architecture.
A benefit of M:N threading is that the Java system can only obtain N pthreads worth of scheduled execution time from the underlying operating system (which is at least better than M:1, as in green-threads). By contrast a 1:1 model (i.e. native-threads) would allow the Java system to swamp the system with runnable threads, crowding out system threads and other Unix applications. Another benefit is that system-wide thread management within the RVM involves synchronizing at most N active threads rather than an unbounded number of threads. For example, a stop-the-world garbage collector [82] merely needs to flag the N currently active threads that they should switch into a collector thread rather than having to stop every thread in the system.
A downside to the M:N threading model is that many native IO operations are blocking operations and the thread invoking the operation will block until the IO operation completes. This causes problems for the JikesRVM

M:N threading model. If a thread invokes such an operation the whole virtual processor will be blocked and unable to schedule any other thread. The JikesRVM attempts to avoid this problem by intercepting blocking IO operations and replacing them with non-blocking operations. The calling thread is then suspended and placed in a IO queue. The scheduler periodically checks pending IO operations and after they complete, the calling thread is moved from the IO queue back into the running queue. The JikesRVM may not always be able to intercept blocking IO operations in native code. As a result a long running or blocked native method can block other threads.

4.2.3 The quasi-preemptive scheduler

As seen in the previous subsection, JikesRVM multiplexes Java threads on virtual processors, implemented as kernel threads. These lightweight threads are neither "run-until-blocked" nor fully preemptive. On the one hand, relying on voluntary yields would be disastrous especially in a server environment, where CPU resources are shared by a large number of threads. On the other hand, arbitrary preemption would have radically complicated the transition to garbage collection and the identification of object references on thread stacks. In JikesRVM a thread can be preempted, but only at predefined **yield points**. By requiring that the points where a thread could lose control of a virtual processor be "safe GC points", JikesRVM can be assured that all threads, not currently running, are stopped at safe GC points. It is only necessary to stop (at safe GC points) those threads currently executing before beginning garbage collection. The compilers will provide location information for object references on a thread's stack at yield points. This allows compilers to optimize code between safe points that would frustrate type-accurate garbage collection [82] if arbitrary preemption were allowed. This *quasi-preemptive* thread switching is implemented in JikesRVM thanks to the aid of the two JIT compilers (described later in details). They generate (as part of every method *prologue, epilogue and loop backedges*) a test of a reserved thread-switch flag. The flag is set by a periodic timer interrupt (see the timer interrupt handler in figure 4.1) and, when the thread reaches a yield point, it checks the thread-switch flag to ask its virtual processor whether it can continue the execution or not. If the virtual processor grants the execution, the thread continues until a new yield point is reached, otherwise it suspends itself so that the virtual processor can execute another thread. In particular, when the thread reaches a certain yield point and the virtual processor informs it that its timeslice has expired, the thread prepares itself to release the scheduler and lets a context switch occur.

Yield points, as it will be clearer in chapter 5, are the ideal candidate as thread "migration points". The execution state in such points is well-known by the JVM, because compilers generated special maps to track object ref-

erences into the thread stack. Moreover, placing yield points in method prologues, epilogues and loop backedges, is way a reasonable tradeoff between performance and "scheduling fairness": thanks to this solution, JikesRVM is able to interrupt Java threads in a timely manner, incurring a nearly negligible overhead [13]. Interrupting instead a native thread in a safe condition is quite an arduous task, because its execution and scheduling is handled by the host operating system and thus with little control from the JVM. Eventually, it is quite straightforwad (as described in chapter 5) to drive the insertion of "migration points" only in those methods where thread migration is considered safe. This allows a tighter control on the granularity of interruption in reactive mobility and confines the possible overhead of reactive thread migration to a very limited set of application methods.

4.3 Baseline or Optimizing? JIT compilation and observability

JikesRVM does not interpret bytecode; rather it compiles each method to machine code and executes the machine code natively. In an adaptive JikesRVM configuration, the **baseline compiler** performs the initial compilation of a method. Methods that are either frequently executed or computationally intensive are identified via a sampling mechanism and recompiled by the **optimizing compiler** [21].
The baseline compiler directly mimics the stack machine behaviour of the JVM [47] specification. It translates bytecodes to machine code quickly, but the resultant machine code typically runs slowly (its quality is competitive with an interpreter). For instance, the simple assignement `a = b + c` gets translated as follows. Local variable `b` is loaded into a register, then pushed on the operands stack (see section 3.1). Local variable `c` is loaded into a register, then push on the stack. The values of `b` and `c` are loaded from the stack back into registers. Their sum is calculated in a register and then stored on the stack. This value is popped from the stack into a register, then stored in local variable `a`.
The optimizing compiler expends more effort to produce high quality machine code for selected methods. This compiler makes much more effective use of registers than the baseline compiler and carries out several state-of-art code optimizations. Its implementation far exceeds the baseline compiler in size and complexity and, for this reason, it deserves a more extensive description, as given in the next subsection. Subsection 4.3.2 then analyses the stack frames produced by the two JikesRVM's compilers, focusing on state observability and capture.

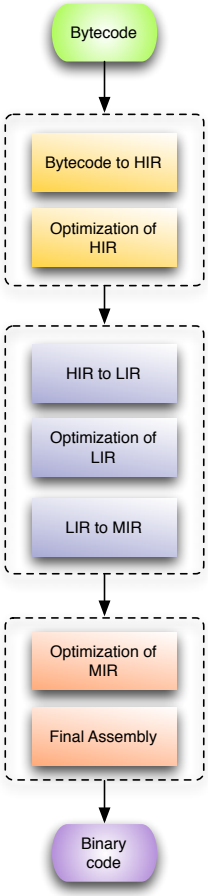


Figure 4.2: Compilation phases in the optimizing JIT compiler

4.3.1 The Optimizing Compiler

Code optimizations are motivated by an emphirical consideration: the bulk of the computation on a Java application will involve only a fraction of the Java source code and, therefore, the optimizing compiler is intended to ensure that these bytecodes are compiled efficiently. The optimizing compiler in JikesRVM is *dynamic* (it compiles methods while an application is running) and *adaptive* (it is invoked automatically on computationally intensive methods). The goal of this compiler is to generate the best possible code for the selected methods on a given compile-time budget. In addition, its optimizations must deliver significant performance improvements while correctly preserving Java's semantics for exceptions, garbage collection and threads.
JIT compilation in JikesRVM's optimizing compiler is a piecemeal process, made up of customizable phases and subsequent optimizations. A schematic representation of these phases is given in figure 4.2, where method bytecodes are transformed three times using an **intermediate representation (IR)** to perform both architecture-independent and architecture-dependent optimizations [2]. Three levels of IR are used:

1. **High Level IR (HIR)**: HIR is architecture independent and resembles the bytecode instruction set, although the IR uses a register transfer language [31] in place of the bytecode stack abstraction. Register-based representations provide greater flexibility for code motion and code transformation than do representations based on trees or stacks. Instructions of the HIR are n-tuples: an *operator* and zero or more *operands*. Most operands represent symbolic and physical registers, memory locations, constants, branch targets or types.

2. **Low Level IR (LIR)**: LIR is architecture independent and resembles the instruction set of a typical RISC machine. LIR expands HIR instructions into operations that are specific to the JikesRVM's object layout and parameter-passing conventions. For example, a virtual method invocation is expressed as a single HIR instruction analogous to the `invokevirtual` bytecode. The single HIR instruction is converted into three LIR instructions that obtain the method dispatch table from an object, obtain the address of the appropriate method body from that table and transfer control to that method body.

3. **Machine Level IR (MIR)**: MIR is architecture-specific; with the exception of a few pseudo-operators that are expanded as part of final assembly, the MIR provides a one-to-one mapping between operators and the target ISA (Instruction Set Architecture).

During the translation from bytecode to HIR, the optimizing compiler constructs exception tables for the method and encodes type information used

in subsequent optimizations. Certain "on the fly" optimizations (e.g. copy propagation, constant propagation, dead-code elimination, etc.) are also performed right after this phase and before the following translation to LIR. In addition, suitably short final or static methods are moved inline.
When LIR is produced, the only optimization performed on it is local common subexpression elimination. Eventually, after low-level optimization, the LIR is converted to the architecture-specific MIR (for either IA32 or PowerPC), and the compiler performs on it optimizations like live variable analysis, branch simplification, peephole optimizations, null check folding and (more importantly) linear-scan global register allocation.
The register allocator maps the infinite set of symbolic registers onto a finite set of physical registers and spill locations (i.e. memory slots allocated in the method frame). The optimizing compiler relies on a variant of the linear-scan register allocation algorithm [56]. In addition, this phase generates prologues, epilogues and calling sequences that respect the JikesRVM and native OS calling conventions. A method prologue allocates the stack frame, saves any nonvolatile registers and checks if a yield has been requested (see *yield points* in subsection 4.2.3). The epilogue checks the yield condition too and then restores any saved registers and deallocates the stack frame. If the method is synchronized, the prologue locks and the epilogue unlocks the indicated object. The final assembly phase of figure 4.2 generates executable machine code from the MIR and finalizes descriptive data structures such as exception tables and GC maps at safe GC points (i.e. yield points).

The whole optimization process is driven by a so-called **compilation plan**. An object of the class `OPT_CompilationPlan` (see figure 4.3) contains all the information necessary to generate machine code for a method. An instance of this class includes, among other fields, the `VM_Method` to be compiled and the array of `OPT_OptimizationPlanElements` which define the compilation steps. The `execute(...)` method of an `OPT_CompilationPlan` invokes the optimizing compiler to generate machine code for the method, executing the compiler phases as listed in the plan's `OPT_OptimizationPlanElements`.
The `OPT_OptimizationPlanner` class defines the standard phases used in a compilation. This class contains a static field, called `masterPlan`, which contains all possible `OPT_OptimizationPlanElements`. The structure of the master plan is a tree. Any element may either be an atomic element (a leaf of the tree), or an aggregate element (an internal node of the tree). Every optimization plan consists of a selection of elements from the master plan; thus two optimization plans associated with different methods will share the same component element objects.

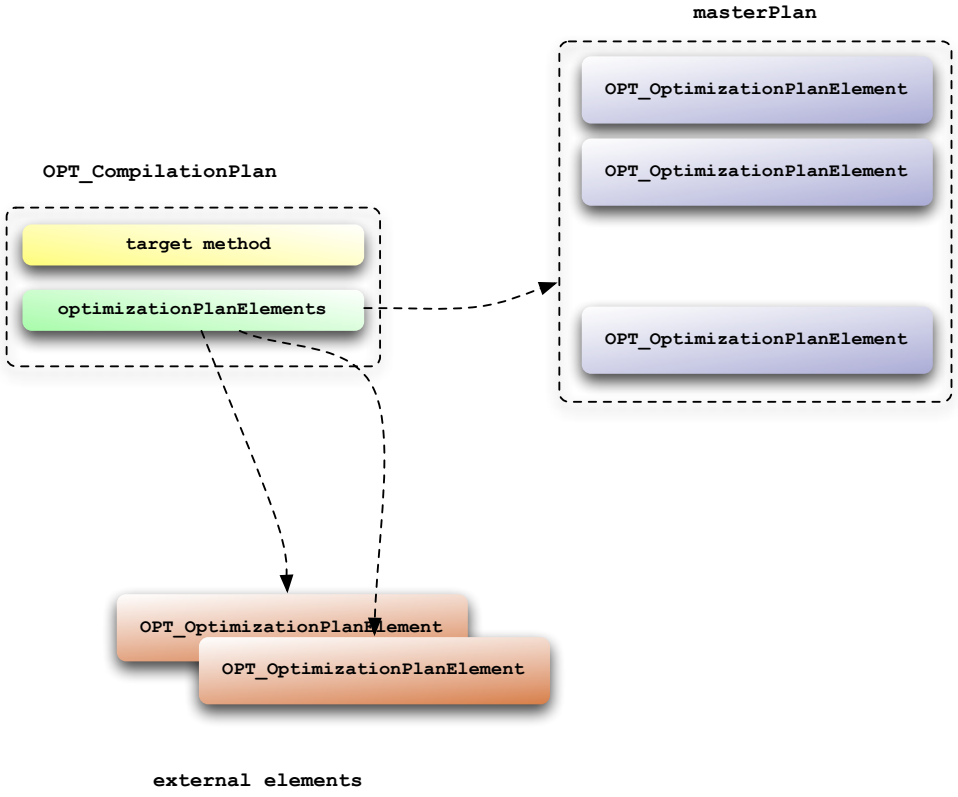


Figure 4.3: Optimization Plans in JikesRVM

4.3.2 Observability of the execution state in baseline and optimizing compilation

The discussion layed out so far on the two JikesRVM compilers (baseline and optimizing) was a necessary premise for some strategic considerations about state observability and capture. These considerations will be essential for a full understading of the the Mobile JikesRVM approach in chapter 5.

JikesRVM register and calling conventions on IA32

Any method (baseline and optimizing compiled) in JikesRVM is called abiding by an architecture-specific **register and calling convention**. Physical registers are divided into *general-purpose* and *floating point* registers. General purpose registers are further subdivided (in an Intel machine) in *volatile* (`EAX, EDX, ECX`) and *nonvolatile* (`EBX, EBP, EDI`) registers. `ESI` and `ESP` are reserved for internal use by JikesRVM.
When the caller method prepares parameters for the callee method (see figure 4.4 or 4.5 later), all parameters that fit are passed in volatile registers. Object references and `int` parameters (or results) consume one volatile register; long parameters, two volatile registers (low-order half in the first); float and double parameters, one floting point register. Parameters are respectively assigned to `EAX` and `EDX` registers. Any additional parameters are passed on the operand stack area of the caller's stack frame. The first spilled parameter occupies the lowest memory slot. Slots are filled in the order that parameters are spilled.
An `int`, or object reference, result is returned in the first volatile register (`EAX`); a float or double result is returned in the first volatile floating point register; a long result is returned in the first two volatile registers (`EAX` and `EDX`, low-order half in the first).
Every kind of frame is prepared during the execution of the method prologue, which is responsible for the following activities:

1. Execute a stackoverflow check, and grow the thread stack if necessary.
2. Save the caller's next instruction pointer (return address in figure 4.4).
3. Store the callee's compiled method ID.
4. Store and update the frame pointer FP to point to the caller's frame.
5. Save any nonvolatile general-purpose registers (GPRS) used by callee.
6. Save any nonvolatile floating-point registers (FPRS) used by callee.
7. Check to see if the Java thread must yield the virtual processor (and yield if threadswitch was requested).

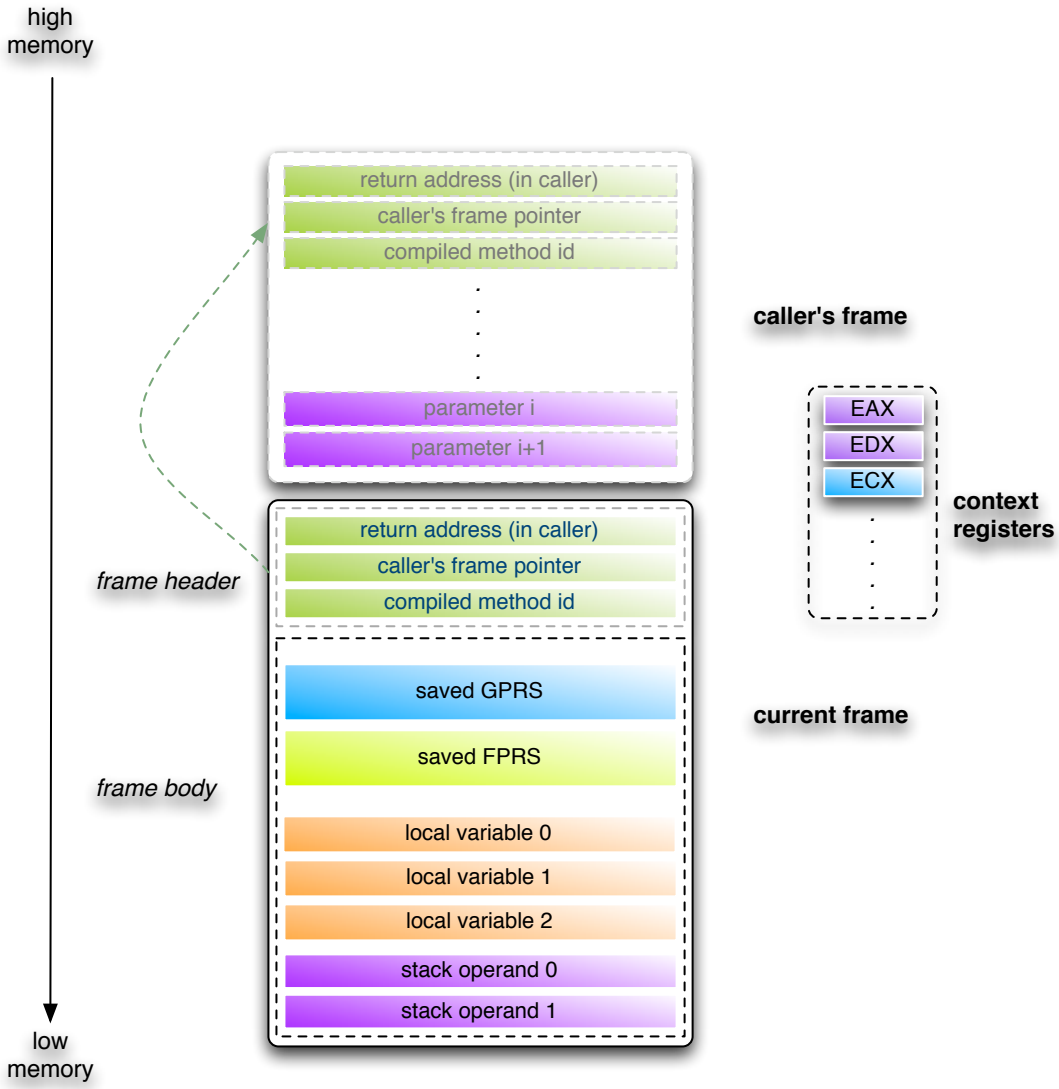


Figure 4.4: Memory layout of a baseline compiled method frame

The epilogue is of course responsible for cleaning up the method state and restoring execution into the caller method, doing the following activities:

1. Check to see if the Java thread must yield the virtual processor (and yield if threadswitch was requested).
2. Restore any nonvolatile floating-point registers used by callee.
3. Restore any nonvolatile general-purpose registers used by callee.
4. Restore FP to point to caller's stack frame.
5. Branch to the return address in caller.

Note that the call stack can only grow in size when a stack overflow occurs, but it is not shrinked when the "overflower" method returns. The reason is that stacks in JikesRVM are Java objects and, as such, they are claimed back by the garbage collector to free the allocated memory.

Frame layout

Every kind of frame starts with a three-slots header (see figure 4.4 or 4.5), which gives some valuable information about the execution state of that method:

- the return address (in the native code) can be translated back to a bytecode index, thanks to compile time maps built by both compilers;
- the frame pointer points to the caller frame and is used to iteratively walk back the stack (from callee to caller);
- the compiled method ID uniquely identifies the called method and can be used to get the method name, signature and class name.

Executing a baseline-compiled method in JikesRVM means running assembly instructions that closely adhere to the stack-based model of the JVM. No optimizations are performed, local variables and stack operands are simply allocated in the method frame and machine context registers are only used as temporary storage for memory operands. Such a deterministic and plain situation results in a simple structure of the method frame in the Java stack. In figure 4.4, the memory layout of a baseline frame is depicted for the Intel32 architecture. A slightly different layout is used on PowerPC, mainly for performance reasons, and the interested reader can refer to the JikesRVM documentation for further details [7].
Figure 4.4 shows how both local variables and stack operands are statically allocated in a baseline method frame. For every bytecode is then possible to precisely locate in memory and extract the value of any local variables or stack operands "live" at that point. The same considerations are no

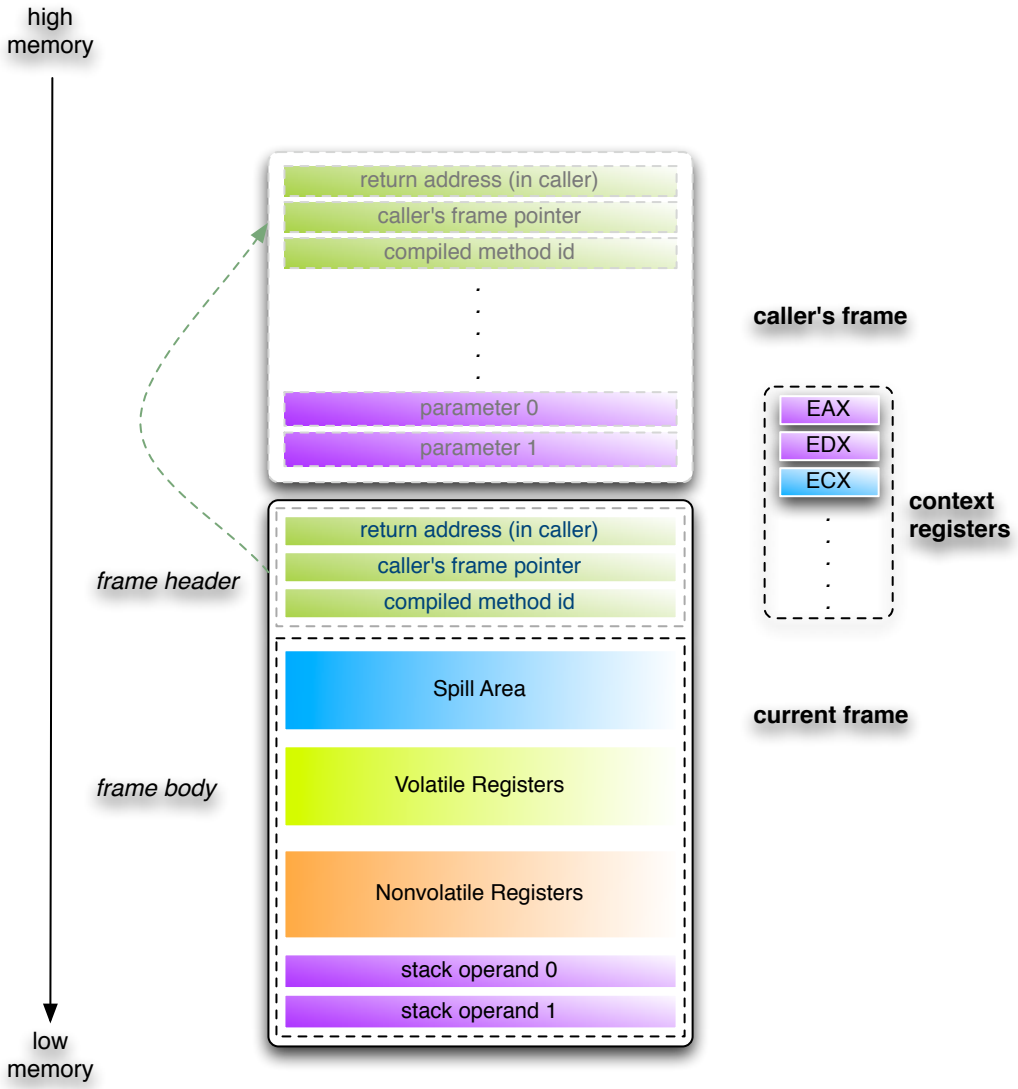


Figure 4.5: Memory layout of an optimized method frame

more valid with an optimizing compiled frame (figure 4.5). First, local variables and stack operands are allocated preferably in context registers, and when no more registers are available, in the spill area of the frame. Yet, **the mapping between locals and stack operands is not known a priori** looking at the frame layout and **it can change at need during the execution of the method**; for example, the optimizing compiler may dynamically decide to allocate a local variable (or a stack operand) in a certain register and then reclaim that register for other purposes. That local variable is spilled to memory at a computed offset in the spill area, thus changing its "physical place". Second, code optimizations can even drop unused variables or encode final values (constants) into assembly instructions. Third, the mapping between optimized frames and logical frames (i.e. Java frames) can be 1:M in case of method inlining: one physical frame includes local variables and stack operands belonging to its called inlined methods.

It emerges clearly how full state observability (as defined in subsection 3.2.2) in optimized frames is way harder to achieve than in baseline frames. This leads to the need for a powerful and low-overhead mechanism (part of the present research work and introduced in chapter 5) to deal with code optimizations and enable state capturing for both baseline and optimized methods.

Chapter 5

The Mobile JikesRVM Framework

Mobile JikesRVM [58] is the name of the thread migration system, implementing the strong mobility approach of the present dissertation work. This chapter fully covers the design and implementation of the system on top of the JikesRVM project, already presented in chapter 4. The presentation of Mobile JikesRVM, as given in the next sections, will follow a "top-down style": the first part of the chapter will analyze Mobile JikesRVM from the programmer's perspective (high-level standpoint), showing how it works and which features it provides to write applications with mobile threads. The second part focuses instead on the architectural perspective (low-level standpoint), presenting the main components of the framework and the JVM techniques implemented to achieve strong thread mobility.

5.1 The "thread serialization" API

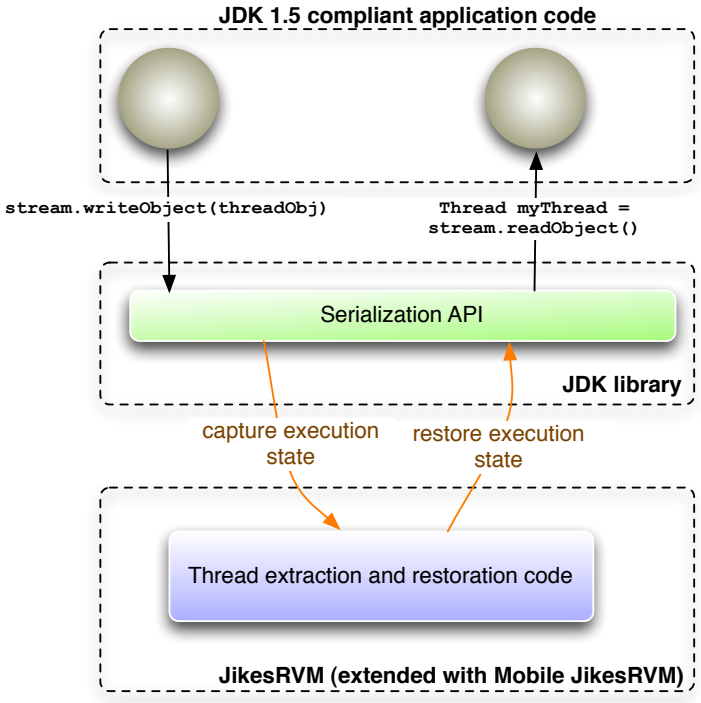


Figure 5.1: Mobile JikesRVM from the programmer's perspective

In subsection 3.3.3, a short anticipation of Mobile JikesRVM has been given, emphasizing the main aspects that make it innovative with respect to other approaches. One heavy criticism to many migration systems has to do with the degree of **usability** of the system. In a few words, every new

migration framework comes with brand new concepts, language constructs or programming constraints that are sometimes uncomfortable and almost never agreed upon. Forcing, for instance, the designer of a distributed application to write her code using the "object-group" concept introduced by the Sumatra [10] migration system will likely be an "unhappy choice" in the long term and (worse) a waste of money.
Starting from these considerations, Mobile JikesRVM has been specifically conceived to **minimize the tools and the know-how needed by the programmer to use it**. The strategical choice has been *working on a well-known and standard technology* (namely the **Java Object Serialization**), in order to extend its scope to thread objects. This means that thread migration boils down to nothing more than "writing a thread in an object stream and reading it back into a (possibly remote) JVM".
Figure 5.1 shows how a regular multi-threaded Java application interacts with the Mobile JikesRVM framework through the serialization API. Thread capture is triggered by writing an object of the `java.lang.Thread` class into an `ObjectOutputStream`. Thread restoration happens automatically when a thread object is read from an `ObjectInputStream` back into the JVM heap. The application code that can use Mobile JikesRVM is any kind of JDK 1.5 compliant code, with no particular structural constraints or special keywords to be used. All the internal thread migration mechanisms are completely hidden to the application, masked under the semantics of the object serialization protocol. As figure 5.1 shows, a thread object is serialized like any other Java object, using the default JDK classes for Object Serialization (package `java.io`). This is perfectly legal, because a thread in Java is first and foremost an `Object`, possibly with fields that can also reference other objects in the heap. The serialization protocol goes recursively into such fields and tries to serialize referenced objects as well. There is only one part that the standard serialization cannot handle and that part is the execution state associated to the thread object, i.e. the call stack.
It is in fact at this point that Mobile JikesRVM comes in, extending the serialization protocol to the call stack (thanks to a special Mobile JikesRVM component described later). The call stack is decomposed in a chain of frame objects and serialized into the underlying stream. The reverse process is carried out at deserialization time, where that same chain is composed back into a "ready-to-go" Java thread. Any errors occured during the thread serialization (or deserialization) process is reported back to the application code throwing a `java.io.IOException` (as any other serialization exception).

```
public class ListenerThread extends Thread {

private int listeningPort;

public ListenerThread(int port) {
    listeningPort = port;
}

public void run() {

    ServerSocket serverSocket = new ServerSocket(listeningPort);

    while(true) {
    try{
        Socket socket;
        socket = serverSocket.accept();

        // Read the object from the socket
        ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
        Object o = ois.readObject();
        if (o instanceof Thread) {
            /* Resume the thread locally */
            Thread newThread = (Thread) o;
            newThread.start();
        }
        socket.close();

        Thread.sleep(1000); // let the thread execute for 1 sec

        // open a client socket to the other host
        socket = new Socket("localhost", 20000);
        ObjectOutputStream oos = new ObjectOutputStream(s.getOutputStream());
        oos.writeObject(newThread);
        socket.close();
    }
    catch(IOException e) { // Serialization Error? }
        ...
    }
}
```

Figure 5.2: The ListenerThread class

5.2 Mobile JikesRVM: an example of use

Starting from the well-know Java Serialization API, programmers can use thread migration in their distributed applications in a seamless and easy way. Before diving into the architectural details of the system, it is useful to show how simple and straightforward can it be to write a Java application that can move threads between two JVMs, using Mobile JikesRVM.
The needed elements are:

1. two running instances of JikesRVM (A and B), with the Mobile JikesRVM extension installed in each of them;
2. a server thread (called *ListenerThread*) on each JikesRVM, listening on a specific TCP port and implementing the migration policy;
3. the application thread that has to be migrated.

A possible `ListenerThread` class is reported in figure 5.2. It is instantiated and started on both JVM A and B and it has a TCP port associated (20000 in the code excerpt). In an infinite loop, the ListenerThread basically

- accepts a socket connection from another host
- reads a thread object from the socket stream
- runs the deserialized thread locally for about 1 second
- opens a client socket with the other host
- serializes back the thread into the new stream

This example of migration policy is a simple infrastructure to move one application thread in a ping-pong style, between two instances of JikesRVM. As it appears evident, the code is plain Java code, using the standard socket API to communicate among different hosts and the serialization primitives `writeObject()` and `readObject()` to migrate threads with Mobile JikesRVM.
Last, it must be pointed out that the bytecode of the application thread (not reported here because not deemed relevant) should be cached in the special *mobile code* directory. This directory is specified by setting the UNIX environment variable `RVM_MOBILECODE`. As anticipated in subsection 3.3.3, only threads whose code is loaded from the mobile code repository are considered migrable by Mobile JikesRVM.

5.2.1 Resource Management in Mobile JikesRVM

The set of all referenced objects of a thread has been previously defined as its data space [34] and, at any point in the execution, is composed of all

the objects that can be reached by the thread through the call stack or its fields. As concerns the stack, the space that the thread is supposed to bring with itself comprises all the objects pointed by parameters, local variables and stack operands of each frame in the stack.

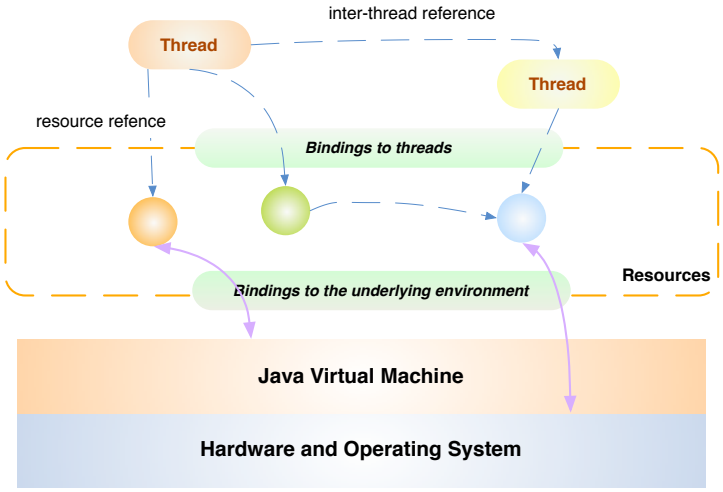


Figure 5.3: Resource relocation in the JVM

Although issues, like resource relocation and binding reconfiguration, pertain more to the application than to the thread migration framework, their importance demands some kind of tool or support, in order to present a coherent set of mobile computing abstractions. In this section, the ideas of **MobileResource**s and **relocation policies** are outlined. A conceptual view of resources and threads is depicted in figure 5.3: Java threads can have references to either active (i.e. other threads) or passive resources (i.e. regular Java objects in the heap). The bindings to the needed resources must be properly rearranged to maintain accessibility and consistency when the computation migrates to new locations. This poses two kinds of problems:

1. *handling the bindings of resources to their underlying execution environment.* This is not a problem if only resources that are not bound to any OS physical entity, like pure Java objects, are considered; on the contrary, resources, such as files, sockets or database objects, cannot be serialized as they are, without carefully managing their binding to the underlying environment.

2. *handling the binding of resources to migratory threads.* [30] identified three typologies for this bindings (by identifier, by value or by type) and proposed some relocation strategies for each of them (by move, by copy, by network reference).

As for the first point, it must be pointed out that moving some resources (e.g. a centralized database) may be neither technically (e.g. the bandwidth is not enough for its size) nor semantically (e.g. it is already in use for queries by other threads) possible. Such issues may be coped with by explicitly introducing the **MobileResource** concept in the programming model and letting the programmer specify the right policy for her resources. Introducing the `MobileResource` entity as an interface, the programmer will be asked to make its resource objects implement such interface, together with a set of useful methods for:

- extracting the resource from its environment in a portable/serializable format (if the resource is fixed an exception will be raised and caught by the framework);
- attaching the resource to the destination environment;
- performing a correct cleanup of the resource, if it is detached from the source JVM (see the proposal by Park and Rice [54])

A simple example of a resource can be that of a `java.io.File` object. A mere serialization of such an object will not produce the actual movement of the underlying file system object. To accomplish this task, the programmer has to introduce its `MovableFile` object, inheriting from `File` and implementing the `MobileResource` interface, with some of the methods detailed above: in particular, calling the "extraction method" will likely return a `byte[]` filled in with the file content; calling the "attach method" will recreate that file in the file system at destination, with its previous content; calling the "detach method" will likely close the file descriptor underneath.
Focusing on the second point above, the problem of the bindings between resources and migratory threads should be addressed. The choice of the right re-binding strategy depends on several factors, from runtime conditions and access-device properties to management requirements and user properties. For instance, a fixed server with no strict constraints on network bandwidth or memory could copy or move the needed resources and work on them locally, whereas a wireless-enabled laptop might want to access that resource remotely without moving it. However, the programming language adopted usually determines the binding strategy. Moreover, the strategy is typically embedded within the mobile application code, thus limiting binding-management flexibility.
The resource management tool of Mobile JikesRVM gives the programmers the means to specify which reference management policy to use, on a per-instance basis. Relocation policies are strategies to adopt when the thread migrates and the framework has to reconfigure all the bindings to its referred objects. Three relocation strategies are allowed:

1. *by copy*: regular objects (i.e. not implementing the `MobileResource` interface) are by default relocated "by copy", i.e. they are serialized

into the object stream and therefore they have to be Java serializable objects. `MobileResource`s must instead implement the inherited `extractState(...)` method, called by the framework to obtain the serializable state of the object to send (e.g. the file content of previous example).

2. *by move*: the object is extracted and serialized as in the "by copy" strategy. Nevertheless, the framework calls the inherited `detach()` method on the resource instance, to let the object carry out clean-up operations [54], such as closing handles on other resources (e.g. calling the `close()` method on a `File` or a `Socket` object).

3. *by ref* (i.e. *by network reference*): the real object is not serialized, but it remains attached to the source environment. A proxy object is instead serialized and it is used at destination exactly as the real object. Field accesses and method calls on such object are transparently forwarded by the framework to the real object on the previous host. More details on this policy are provided later in this chapter.

```
public class MyThread1 {
     private MyResource r;
     private MyThread2 t;
     public void run(){
          r = new MyResource();
          t.setResource(r);
          try{ setPolicy(r,BY_REF);
          // …
          migrate(host,port);
          // resumed --> uses the resource remotely
          r.field1="Hello"; } catch(…){…}
          // …
     }
}
```

```
public class MyThread2 {
     public int setResource(MyResource r){
          MyResource r1 = r;
          try {setPolicy(r1, BY_MOVE);} catch(..){…}
          // perform some operation with the resource
     }
}
```

Figure 5.4: Resource policies: an usage example

As shown in figure 5.4, each thread can register its policies on its resources (by means of `MobileThread.setPolicy()` method) and these policies are local to the registering thread. These registrations can be modified (a new registration on the same object overwrites the old one) throughout the life of the thread, depending on the needs of the application. When the thread

decides to move, Mobile JikesRVM asks the resource management tool to apply the registered policy on each referenced resource and it obtains the serializable representation for that resource. Once at destination, the thread is deserialized and every resource is reconnected according to the chosen policy.

5.3 An overview of the framework

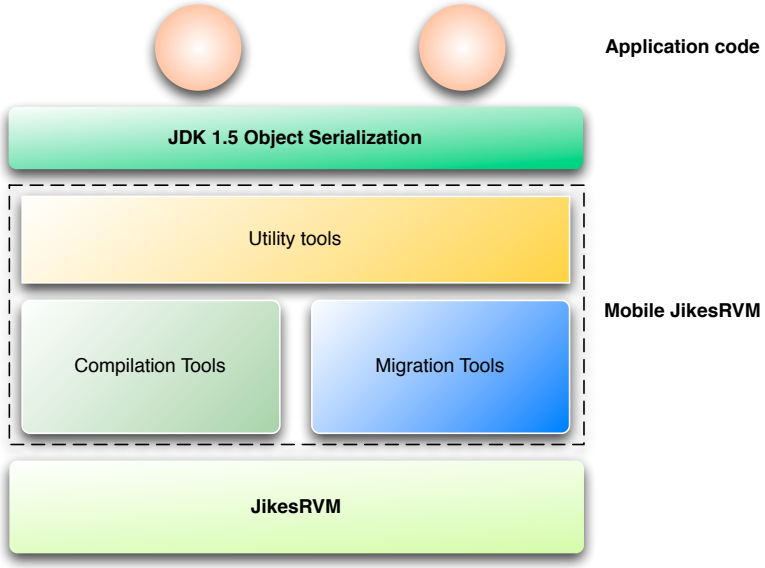


Figure 5.5: The three toolsets of Mobile JikesRVM

From an architectural standpoint, Mobile JikesRVM has been organized into three sets of tools, each one being the subject of the next three sections of the chapter:

- the **compilation tools**, i.e. a set of two special JIT compilers, designed to produce suitable code that can be fully migrated;
- the **migration tools**, comprising components essential to perform stack capturing and restoration when migration happens.
- the **utility tools**, which include some additional components for the programmer to deal with resource (i.e. dataspace) relocation. It shows how the ideas of subsection 5.2.1 have been implemented in Mobile JikesRVM.

5.3.1 The Compilation Tools: special baseline and optimizing compilation

Two "migration-friendly" JIT compilers (a baseline and an optimizing) are the main components of the Compilation Tools. They live aside the default JikesRVM compilers and should be activated only when a dedicated compilation is needed. In order to decide "when" they should be enabled, it comes in handy to introduce here the key concept of **Mobile Method**, before analyzing the design of the two compilers.

The Mobile Method concept

Java classloaders are key JVM components to implement *dynamic class loading* [47], making it possible to fetch the `.class` file containing the metadata (and the bytecode) of a needed class. Every method of a loaded class is usually compiled into native code right before its first invocation by the JIT compiler. The codebase (i.e. the physical location) from which the class file is fetched has nearly no influence on the kind of compilation performed on its methods. Except for untrusted bytecode coming from external URLs, which demands extra verification and security checks, the produced machine code is always the outcome of the same JIT compilation process.
But when it comes to mobile code, it is terribly important to carefully compile methods that are expected to be invoked by mobile threads. In subsection 3.2.2, it has been pointed out that code produced by plain JIT compilation may not always determine a fully observable execution state. This point has been further explored in subsection 4.3.2, where the physical structure of the method frame in JikesRVM has been discussed for both the baseline and the optimizing compiler. The conclusion is crystal clear: JIT compilation and code optimizations are obstacles to a full observability of stack frames, because they willingly violate the stack-based architecture of the JVM in favor of better code quality and execution speed.
Leveraging the power of JIT compilation without hampering performance has been among the top priorities in the design of Mobile JikesRVM. A lightweight state tracking technique (described later in this section) enables full observability of method frames at migration points, with very little performance loss of the produced code. Although acceptable for application code that is supposed to be migrated, there is no valid reason why this penalization should involve non mobile code as well (e.g. third-party libraries or middleware code). For this reason, Mobile JikesRVM chose to strongly confine the effect of "migration-friendly" compilation to a limited subset of application methods, using the classloading codebase to mark a new kind of class methods: **Mobile Methods**. The codebase directory is defined by the value of the `RVM_MOBILECODE` environment variable, set and exported before starting JikesRVM up. The rule is that

every method whose bytecode is loaded directly from the `RVM_MOBILECODE` directory, or from any jar file located in the `RVM_MOBILECODE` directory, is marked as a Mobile Method

```
static VM_Method readMethod(VM_TypeReference declaringClass, ...) {
    ...
    // here, make the choice between VM_NormalMethod and VM_MobileMethod
    if (canbeMobileMethod(memRef, declaringClass, tmp_exceptionTypes)==true)
        method = new VM_MobileMethod(...);
    else
        method = new VM_NormalMethod(...);
      ...
}
```

```
static boolean canbeMobileMethod(...) {
    if ((memRef.getName() == VM_ClassLoader.StandardObjectInitializerMethodName)||
        (memRef.getName() == VM_ClassLoader.StandardObjectInitializerHelperMethodName)||
        (memRef.getName() == VM_ClassLoader.StandardClassInitializerMethodName))
        return false;

    if(exceptionTypes!=null) {
        for(int i=0; i<exceptionTypes.length; i++)
        if(exceptionTypes[i] == JavaIONotSerializableException)
            return false;
    }

    if (mobileCodeRepository == null)
        mobileCodeRepository = System.getenv("RVM_MOBILECODE");

    String className = null;
    if(mobileCodeRepository!=null) {
        ClassLoader cl = declaringClass.getClassLoader();
        if (cl instanceof java.net.URLClassLoader) {
            className=declaringClass.getName().classFileNameFromDescriptor();
            if(cl.getResource(className).toString().indexOf(mobileCodeRepository)!=-1)
                return true;
        }
    }
    return false;
}
```

Figure 5.6: An excerpt from `com.ibm.JikesRVM.VM_Method.java`, showing the instantiation of Mobile Methods

Introducing Mobile Methods in JikesRVM has implied, first of all, subclassing the `VM_NormalMethod` class (from the `com.ibm.JikesRVM.classloader` package) with a new `VM_MobileMethod` class. Instances of this subclass are created to represent only those methods following the above rule. The code that implements this concept is reported in figure 5.6. The shown `readMethod(...)` is executed at class loading time, when JikesRVM discovers fields and methods of the class. `canBeMobileMethod(...)` is called by the previous `readMethod(...)` to decide whether or not the method being loaded will be a Mobile Method. Some special methods (like class initializers and object constructors) are immediately excluded because migrating a thread while in the process of initializing an object or a class has

little meaning.
One clear advantage of the Mobile Method concept is its **ease of use** for the application programmer. No special syntax or keywords (like the `migratory` marker used by **JavaGo** [62]) are needed to make a regular Java thread a "mobile thread", i.e. fully and strongly migrable computation flow.

- Setting the `RVM_MOBILECODE` variable
- and downloading the application code in that directory

are the two only steps needed to exploit strong migration through Mobile JikesRVM (no need to rewrite or adapt legacy Java code!). Furthermore, while every method loaded from the mobile code repository is automatically marked as a Mobile Method, the programmer has still the possibility to prevent certain methods from being marked as such. Declaring a method as throwing `java.io.NotSerializableException` forces Mobile JikesRVM to consider it as a regular non mobile method.

Activating "migration friendly" compilation of a Mobile Method

The migration framework selects Mobile Methods at classloading time, according to their original codebase. Yet, such methods are like any other method in the JVM from the compilation point of view. In JikesRVM, JIT compilation is triggered dynamically in two situations:

1. the first time the method is invoked (baseline compilation);
2. when the method is selected to be recompiled (optimizing compilation).

In order to activate the special "migration friendly" compilation on Mobile Methods, Mobile JikesRVM has slightly modified the main compilation loop in the JVM, as shown in figure 5.7. The first time a method is compiled the `baselineCompile(...)` method is called by the compilation thread and, instead of being compiled by the default `VM_BaselineCompiler`, a Mobile Method is compiled by the `VM_MobileCompiler`. The latter compiler is almost identical to the default baseline compiler, except for the insertion of migration points instead of regular yield points (see subsection 5.3.2).
When the optimization system of JikesRVM [15] decides that the method is worth being optimized, it calls the `optCompile(...)` method of figure 5.7, passing the method object and the compilation plan. Mobile Methods are compiled by the optimizing compiler using a special `mobileOptimizationPlan` instead of the default optimization plan.

```
public static VM_CompiledMethod baselineCompile(VM_NormalMethod method) {

    VM_CompiledMethod cm=null;
    VM_Thread currentThread = VM_Thread.getCurrentThread();

    ...

    if ((VM.runningVM) && (method instanceof VM_MobileMethod)) {
         cm = (VM_BaselineCompiledMethod)
             VM_CompiledMethods.createCompiledMethod(method, VM_CompiledMethod.BASELINE);
             new VM_MobileCompiler((VM_BaselineCompiledMethod)cm).compile();
    }
    else
         cm = VM_BaselineCompiler.compile(method);

    ...

    return cm;
}
```

```
private static VM_CompiledMethod optCompile(VM_NormalMethod method,
                                            OPT_CompilationPlan plan){

   ...
   if(method instanceof VM_MobileMethod)
       plan.optimizationPlan=mobileOptimizationPlan;

   VM_CompiledMethod cm = OPT_Compiler.compile(plan);

   ....
   return cm;
 }
```

Figure 5.7: An excerpt from `com.ibm.JikesRVM.VM_RuntimeCompiler.java`, showing the activation of mobile compilers

Describing the Mobile Optimization Plan

As introduced in section 4.3.1, the optimization of a method in JikesRVM consists of a series of compiler phases performed on the method. These phases transform the IR (intermediate representation) from bytecodes through HIR (high-level intermediate representation), LIR (low-level intermediate representation), and MIR (machine intermediate representation) and finally into machine code. Various optimizing transformations are performed at each level of IR and these transformations are driven by a specific optimization plan.

Mobile JikesRVM constructs (at the JVM boot) a specific optimization plan for Mobile Methods (called **Mobile Optimization Plan**), by including all the `OPT_OptimizationPlanElement`s contained in the master plan which are appropriate for this compilation instance. Furthermore, it includes other elements whose aim is to enable full state observability even in presence of code optimizations. Those added `OPT_OptimizationPlanElement`s will all work on inserting and maintaining, throughout the whole plan, a special IR instruction, called `OSR_Point`.

Inserting OSR_Points into HIR During the translation from bytecode to HIR, the basic idea is to abstractly interpret the bytecode stream, translating it into a register-based IR along the way. At each program point, this phase keeps an abstract stack and an abstract local variable array. The approach followed by Mobile JikesRVM is to save this state before each one of the following "call" bytecodes:

- `invokevirtual`
- `invokespecial`
- `invokestatic`
- `invokeinterface`

The state is saved filling in a special IR instruction built-in in JikesRVM for other purposes: it is the **OSR_Point** instruction, used by the OSR (On-Stack Replacement) subsystem [32]. OSR is a tool used to achieve on-demand recompilation of methods in the call stack. Method recompilation allows JikesRVM to replace on the fly a baseline method activation (on top of the stack) with its optimized version. As concerns the optimizing compiler, OSR is used only for a single purpose: *enabling guarded inlining of virtual method calls*. In a few words, while static and final method calls can be safely inlined into the caller method, this is not true for virtual methods. Such methods are linked at runtime, due to the polymorphism in the Java language. Yet, in some particular cases, the target method of a virtual call

can be estimated with a high probability at compilation time, if certain assumptions are valid. Guarded inlining means producing an inlined version of the method that can be invalidated at runtime, if the inlining assumptions fail. **OSR_Points** allow the OSR subsystem to reconstruct a non-inlined version of the method frame and continue execution with that version.

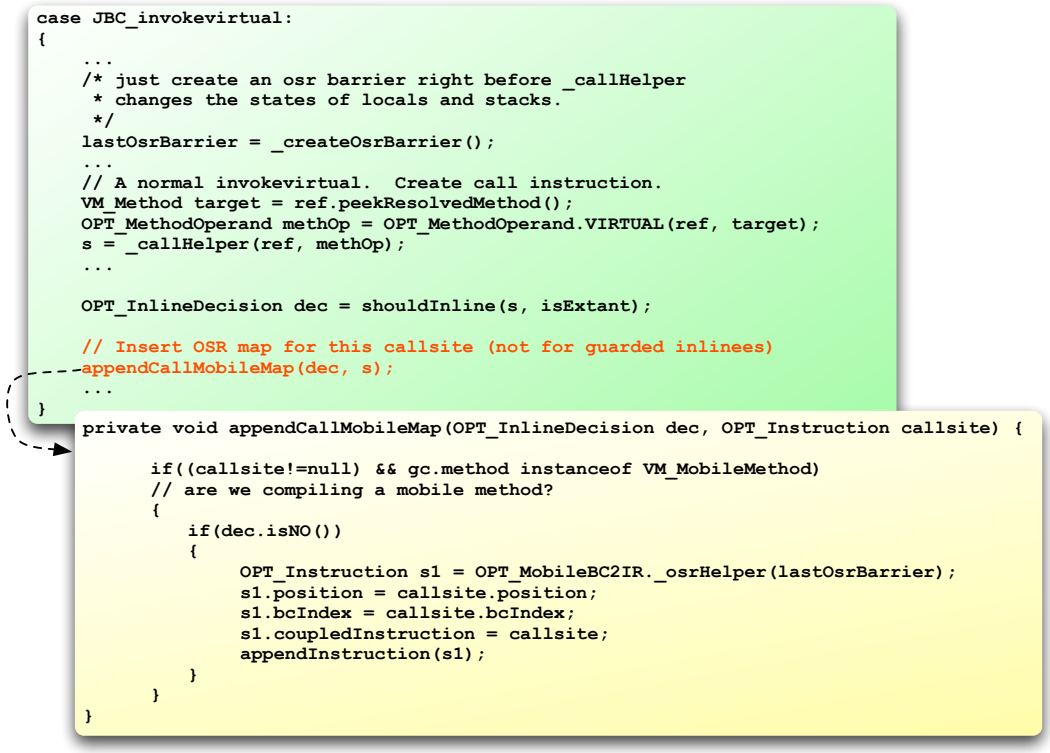


Figure 5.8: OSR_Point construction during the bytecode to HIR translation phase

The process of the `OSR_Point` construction and insertion is shown in figure 5.8, only for an `invokevirtual` bytecode. Right before translating the `invokevirtual` with an HIR `call` instruction, Mobile JikesRVM creates an *osr barrier*, which contains a snapshot of locals and stack operands at that point. This snapshot is then stored in the corresponding `OSR_Point` instruction, to be passed and updated by other compilation phases. The `OSR_Point` instruction is inserted in the instruction flow right before the HIR `call` (see `appendCallMobileMap(...)` in figure 5.8). It must be observed that when the target method of the call is an inlinee method, the `OSR_Point` instruction is not inserted, but the corresponding osr barrier is "merged into" the osr barrier of the caller method. This merge process is repeated recursively

for each level of inlining (i.e. initial method – inlinee1 – inlinee2 – ... – leafMethod), until the last non inlinee method is reached. At that point, the `OSR_Point` instruction is inserted right before the final call, along with its cumulative osr barrier. **This trick allows Mobile JikesRVM to recover the execution state of a method, even in case of multiple inlined methods**, because the state of each inlinee in the call chain is saved into its osr barrier and can thus be extracted back from the caller frame.

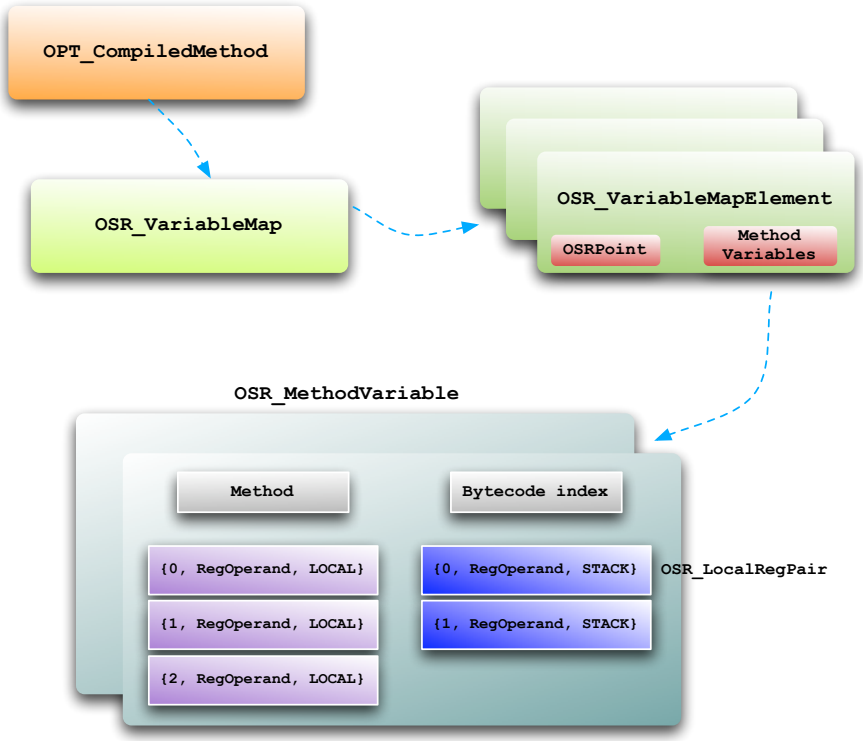


Figure 5.9: Organization of OSR maps in Mobile JikesRVM

OSR_Points throughout the optimization phases The HIR code with its inserted `OSR_Points` put before method calls traverses the other compilation phases of the mobile optimization plan. Like a `call` instruction, an `OSR_Point` will constrain some optimizations, including dead code elimination, load elimination, store elimination, and code motion. In its original design, an `OSR_Point` instruction transfers control to the exit block, so there is no merge back to reachable code after it (it is used to trigger recompilation of failed guarded inlining). Mobile JikesRVM changes this aspect, so that `OSR_Points` inserted by the mobile optimization plan do not

branch to the exit block, but simply come before their associated call instruction.

During the *Live Analysis* compilation phase, the optimizing compiler performs a flow-sensitive iterative live variable analysis. The result of this analysis is live ranges for each basic block and this information is vital to update the state captured in the various `OSR_Point`s of the IR. For this reason, during this phase live ranges information are used to update locals and stack operands in each `OSR_Point` of the method.
Furthermore, during this phase `OSR_Point`s are used by Mobile JikesRVM to build **OSR maps** for the method. These maps are organized as depicted in figure 5.9. For each optimizing compiled method, Mobile JikesRVM prepares an `OSR_VariableMap`, pointing to a linked list of `OSR_VariableMapElement`s. There is one `OSR_VariableMapElement` for each `OSR_Point` instruction in the IR code. Considering that each `OSR_Point` may contain the state of a chain of inlined methods (i.e. caller - callee - callee ...), an `OSR_VariableMapElement` can have one or more `OSR_MethodVariable` elements for each inlined method. Then, each `OSR_MethodVariable` contains the variables of the method at a certain bytecode index (e.g. the one of the related `invokevirtual` instruction). Each variable is represented as a `OSR_LocalRegPair` element, with the information about its type (i.e. LOCAL or STACK), index and a reference to the associated `OPT_RegisterOperand` telling to what symbolic register the variable has been allocated.

From OSR maps to physical frame maps During the initial compilation phases of the mobile optimization plan, variables and stack operands are allocated into symbolic registers of the IR register-based language. The number of symbolic registers is infinite, but the number of physical register is not. For this reason, the optimizing compiler executes (as one of its last phases) a so-called **linear-scan register allocator** [56] algorithm to map symbolic registers to physical locations in the JVM. Two kinds of physical locations are available:

- physical machine registers
- frame memory spills

Physical registers are very limited on Intel processors and are therefore allocated carefully to the most used variables of the method. For every other remaining variable (or stack operand), an indexed slot in the method frame (called *spill*) is instead reserved, as depicted in figure 4.5. After the register allocation phase, the osr map of the method must be updated, replacing symbolic registers with physical registers or spill offsets. Nonetheless, physical registers are overwritten across method calls, because they are extensively reused. In order to restore the values of registers at state capturing time,

Mobile JikesRVM takes a snapshot of their value right before executing the `call` of the callee method. The snapshot is saved into a special register-save area reserved in the optimizing frame, so that the frame extractor (presented later in section 5.3.2) can recover the saved registers and use their value.

When the MIR of the method is finally converted to machine code, the machine code offset of each instruction is known and this offset is extremely important to find the right osr map during state capturing (see section 5.3.2). In Mobile JikesRVM, osr maps are in fact indexed by means of the machine code offsets of their coupled `call` instructions, so that they can be easily found at execution time. Therefore, in this last phase of the compilation the osr map is updated with the right machine code offsets and then translated into its final form: an `OSR_EncodedMap`. The `OSR_EncodedMap` is basically a compact string of bits that stores all the information previously contained in the osr map. This representation allows saving a lot of space for the frame state maps in each optimized method.

5.3.2 The Migration Tools: frame extraction and installation

The Migration Tools package contains classes needed to extract the call stack of a thread at capturing time, to transform it in a portable form and to rebuild the stack again at destination. Thread migration, as already pointed out, can be **proactive** (i.e. voluntarily triggered by the target thread) or **reactive** (i.e. notified by another thread). Later in this section, the two migration tools (**FrameExtractor** and **FrameInstaller**) are explained. Such important tools have been implemented and used in Mobile JikesRVM to achieve both proactive migration and reactive migration. However, in order to fully grasp the dynamics of the migration process (especially the more complex reactive case), a new concept must be introduced and commented soon: the **migration point**.

Reactive Mobility through Migration Points

Born as way to decide the most suitable kind of JIT compilation, the Mobile Method concept comes in handy for another reason: selecting which methods should be interrupted by a *reactive migration* request. As subsection 3.2.1 has pointed out, choosing the right **granularity of interruption** for the target thread helps reducing potential inconsistency risks, e.g. caused by an undesired migration occurred during a critical I/O operation. Allowing reactive migration only during the execution of a Mobile Method can be a fair approach if the following assumption is made:

the logic of a migrable computation is usually coded separately from the set of libraries and externals tools used by it.

In other words, the code of the mobile application is implemented in one or more classes (e.g. extending `java.lang.Thread`) and the rest of the code is made up of referenced classes, like graphical packages, mathematical libraries or communication utilities. Reviewing some well-known distributed and parallel computing applications, it emerges that libraries referenced by the main application logic exhibit many of the following features:

- they carry out operations that are of a *local nature* (e.g. a graphical function can be elaborating on a local videocamera stream);
- they use *low-level or heavy objects that are almost always not serializable* (e.g. database objects);
- they perform *non interruptable critical functions* (e.g. interacting with physical devices), often invoking native C/C++ libraries.

Starting from these rules of thumb, only Mobile Methods (i.e. bytecode loaded from the mobile code repository) can be interrupted by a reactive migration request, in a clearly identifiable number of points, called **migration points**.

Inserting migration points

```
private final void genMobileMethodSwitchTest(int whereFrom) {
    if (!isInterruptible) {
        return;
    }

    VM_ProcessorLocalState.emitMoveFieldToReg(asm, S0,
                          VM_Entrypoints.activeThreadField.getOffset());
    asm.emitMOV_Reg_RegDisp(S0, S0,
                   VM_Entrypoints.takeMigrationPointField.getOffset());

    asm.emitADD_Reg_RegDisp(S0, VM_RegisterConstants.ESI,
                        VM_Entrypoints.takeYieldpointField.getOffset());
    asm.emitCMP_Reg_Imm(S0, 0);

    VM_ForwardReference fr1;
    fr1 = asm.forwardJcc(VM_Assembler.EQ);

    if (whereFrom == VM_Thread.PROLOGUE) {
        asm.emitCALL_RegDisp(JTOC, VM_Entrypoints.mobilePrologueMethod.getOffset());
    } else if (whereFrom == VM_Thread.BACKEDGE) {
        asm.emitCALL_RegDisp(JTOC, VM_Entrypoints.mobileBackedgeMethod.getOffset());
    } else { // EPILOGUE
        asm.emitCALL_RegDisp(JTOC, VM_Entrypoints.mobileEpilogueMethod.getOffset());
    }
    fr1.resolve(asm);
    ...
}
```

Figure 5.10: Code excerpt that inserts migration points in a Mobile Method

Migration points in Mobile JikesRVM are implemented as a subset of the JikesRVM built-in yield points (described in subsection 4.2.3). Yield points are a couple of hidden assembly instructions, inserted by the JIT compiler in method prologues, epilogues and loop backedges. They force the running thread to release the control of the virtual processor if a thread-switch has been requested (i.e. the timeslice has elapsed). Migration points are basically yield points inserted in Mobile Methods and, as such, they serve as

- scheduling points, if the timeslice for the thread has expired;
- migration points, when a reactive migration has been notified to the thread.

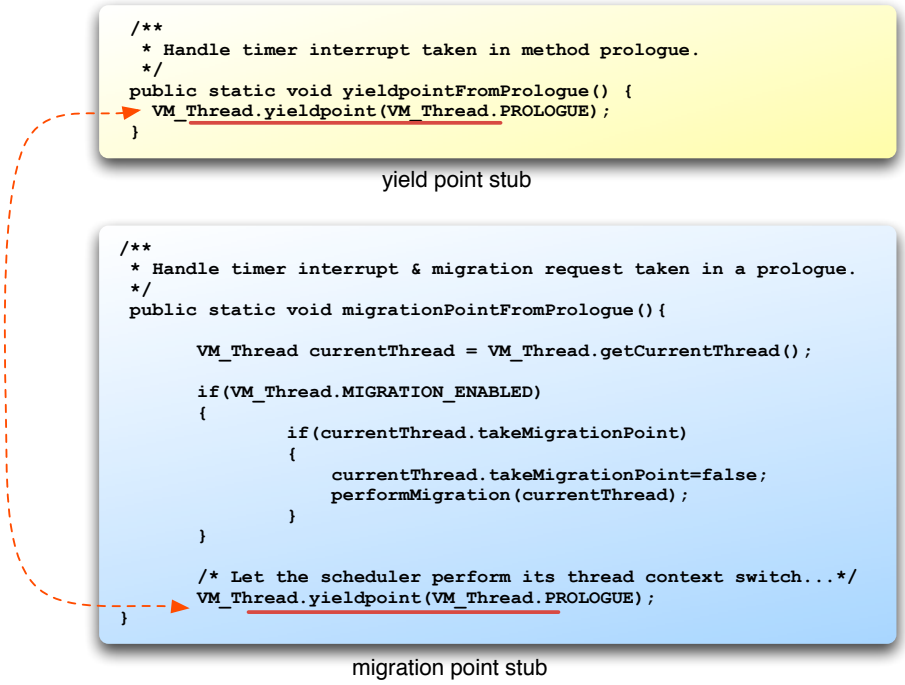


Figure 5.11: A migration point stub compared to a standard JikesRVM yield point

The code excerpt of figure 5.10 shows the code that is in charge of generating assembly instructions to check for a pending thread switch or migration request. The `genMobileMethodSwitchTest(...)` method is called at every prologue, epilogue and loop backedge. A pending thread switch request is represented by a non-zero value of the `takeYieldpoint` field in the current `VM_Processor` object (i.e. the JikesRVM virtual processor). Mobile

JikesRVM introduces another flag to signal thread migration on a target thread, and that flag is stored in the `takeMigrationPoint` field of the current thread object. A migration point is thus "taken" if either of the above two flags is set (in the code of figure 5.10, this means that the sum of `takeMigrationPoint` plus the value of `takeYieldpoint` is non zero). And "taking a migration point" means nothing more than calling a special stub method that performs the proper thread-switch or migration actions. Three methods have been devised (for the prologue, epilogue and loop backedge) and the code of the prologue version is reported in figure 5.11, compared to a standard JikesRVM yield point.
Migration is initiated only if the `takeMigrationPoint` flag of the current thread is true and migration is enabled in the system. The referenced method `performMigration(...)` is the one doing the actual work of state capturing and it will be explained in the next subsection, along with the whole reactive migration process. Please, note from the red lines in figure 5.11 how a migration point subsumes a yield point in a Mobile Method, because it executes the same scheduling routine when just a thread switch has been requested on the current thread. Furthermore, having a migration point instead of a default yield point has a negligible impact on the execution speed of the compiled method, because it only adds two more comparison instructions (the two nested `if` of figure 5.11). This emphirical consideration will be confirmed by the performance evaluation tests reported in chapter 6.

The reactive migration process at a glance

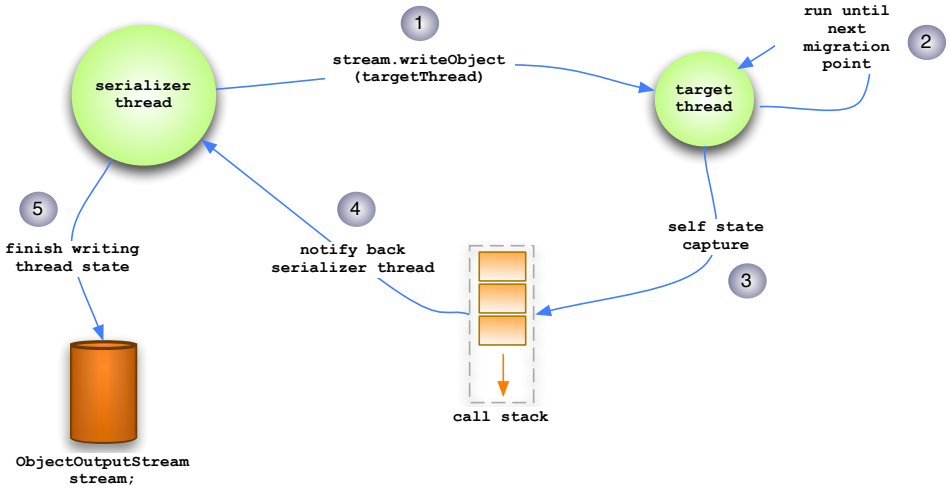


Figure 5.12: The reactive migration process

Migration points have been exploited in Mobile JikesRVM to implement reactive thread mobility. The fundamental steps comprising the reactive migration process are enumerated in figure 5.12.

step 1 First, an external thread, called *serializer thread*, opens an `ObjectOutputStream` (from the `java.io` package) to serialize the *target thread*. The `java.lang.Thread` class is considered by default not serializable. Mobile JikesRVM makes it implement `java.io.Serializable` and adds two methods to it to implement the custom thread serialization protocol: `writeObject(...)` and `readObject(...)` are called by the serialization system respectively when an object of type `Thread` is about to be serialized into an `ObjectOutputStream` or deserialized from an `ObjectInputStream`.
Figure 5.13 shows how the custom serialization protocol works for the `Thread` class. It first checks if the target thread has been started or is already dead and, in this case, reverts to a plain serialization of the thread object and its fields. Yet, the most interesting scenario is when the target thread is still running. The serializer thread cannot write its state until the target thread has stopped to a migration point and captured the state. To this purpose the serializer is temporarily suspended on a synchronization object, waiting for a `notify(...)` to wake it up and start the serialization.

step 2 The target thread runs normally through its code and, when it reaches a migration point (method prologue, epilogue or loop backedge), it checks if a migration request has been notified by another thread. If so, it calls the method stub of figure 5.11, to start the self-capturing process of its execution state.

step 3 As shown in figure 5.11, if `takeMigrationPoint` is set to true, thread migration is started by the target thread itself through the aforementioned `performMigration(...)` method. Figure 5.14 reports an excerpt from its source code. State capturing is done through the depicted `collectFrames(...)` method. This method performs a stack "self-walkback" from the last executed method till the bottom of the call stack, by means of the **Frame Extractor** tool, described later in section. The Frame Extractor, after being initialized with the thread to inspect, must be invoked like an iterator and, at each invocation, it returns the state of the current frame extracted in the form of a **VM_MobileFrame** object (presented later). Frame by frame, the call stack is traversed completely and translated into a linked list (*chain*) of `VM_MobileFrame`s. If any error invalidates the capturing process, the Frame Extractor throws an `IOException` and the process is aborted.

Figure 5.13: The custom serialization method in the `java.lang.Thread`

Figure 5.14: The method that triggers state capture in `java.lang.Thread`

step 4 Either when state capture completes or if something goes wrong, the serializer thread is notified by the target thread and can thus continue the serialization. The target thread (as shown in figure 5.14) then kills itself throwing a `ThreadCapturedException`.

step 5 If the call stack has been successfully captured and converted in a chain of `VM_MobileFrame` objects, the whole thread object (with its fields) and the chain of frames can be serialized into the stream. At this point it can happen that one of the objects referenced by the target thread (and captured in a `VM_MobileFrame`) may not be serializable. This causes an abrupt stop of the serialization process and an NotSerializableException is thrown back to the serializer thread, as it happens normally with any object serialization.

Thread deserialization at a glance

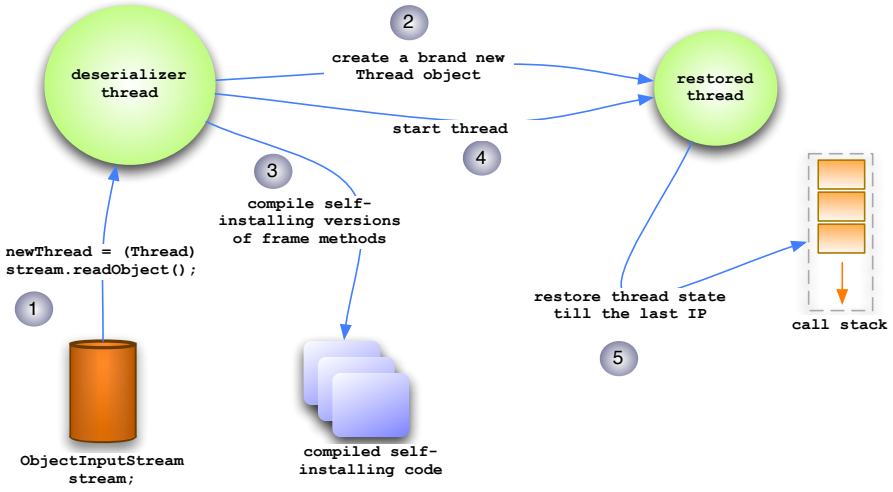


Figure 5.15: The deserialization process in Mobile JikesRVM

Rebuilding a thread in Mobile JikesRVM involves two actors (the *deserializer thread* and the *restored thread*) and a series of sequential steps, as depicted in figure 5.15. The code that implements step 2 and 3 is also reported in figure 5.16.

step 1 The deserializer thread invokes the `readObject(...)` method on an open `ObjectInputStream`. In this phase the bytecode of the thread class is loaded and initialized (if not yet done).

```
private void readObject(ObjectInputStream in) throws IOException {
      frames = null;

      // Read the Thread object with all its subfields
      initThreadObject();

      in.defaultReadObject();
      numFrames = in.readInt();
      if(numFrames > 0){
          frames = new VM_MobileFrame(numFrames);
          frames.readExternal(in);
      }

      // If the thread has a captured execution state..then install it
      if(frames != null) {
          takeSpecialRun = true;

          VM_FrameInstaller fi = new VM_FrameInstaller(this);
          fi.compileFrameMethods(in);

          frames = null; // now the frames chain is no longer needed!!!
          numFrames = 0;
      }

      // We don't call start() on the rebuilt thread,
      // because the programmer may not want to
      // immediately start the deserialized thread.
  }
```

Figure 5.16: The `readObject(...)` method implementing the custom deserialization protocol

step 2 A new `Thread` object is created and the `readObject(...)` method is called upon it. Every field of the thread object is deserialized and last, but not least, the chain of `VM_MobileFrame`s is rebuilt in the heap. In this phase, every object referenced by the thread is reconstructed in memory, before being used again by the restored thread.

step 3 Given the chain of frames successfully deserialized with the thread object, Mobile JikesRVM relies upon the other migration tool, called **Frame Installer**, to re-install the execution state into the newly allocated call stack. The latter operation requires precompiling each frame method, from the `Thread.run(...)` method to the last captured one, in a special *self-installing form* (see the FrameInstaller subsection later for this compilation technique).

step 4 In order to resume the execution of the restored thread from the last executed instruction, the deserializer thread must call the `Thread.start(...)` method on it. This can be done in every moment and until that moment the restored thread remains in a sort of embryonic form.

step 5 Once restarted, the restored thread quickly rebuilds the execution state, pushing every piece of each frame back into the call stack and eventually jumping to the next instruction address.

Moving method state in a portable form: `VM_MobileFrame`

In section 3.1, it has been stated that the JVM is a stack-based machine and, as such, it pushes a *Java frame* on top of the Java stack for each called method. Java frames are just a reference abstraction that JVM designers have to implement, without any constraints about the physical memory layout of the frame. JVMs running on PowerPC architectures (e.g. JikesRVM itself) will, for instance, take advantage of the big number of machine registers and will try to optimize the structure of the frame accordingly.
Mobile JikesRVM relies upon the **FrameExtractor** component, presented later, to translate the physical state of methods in the call stack to a set of Java frames. The structure of the Java frame in Mobile JikesRVM is reported in the code excerpt of figure 5.17. An instance of `VM_MobileFrame` contains all the necessary information needed to reconstruct the execution of the method in another JVM. Physical frames contain usually VM-dependent stuff, like stack pointers or cached registers. Yet, this stuff has no correspondent at the bytecode level. It is therefore responsibility of the frame extraction tool to browse through these data and use them to properly rebuild the original bytecode-level state.

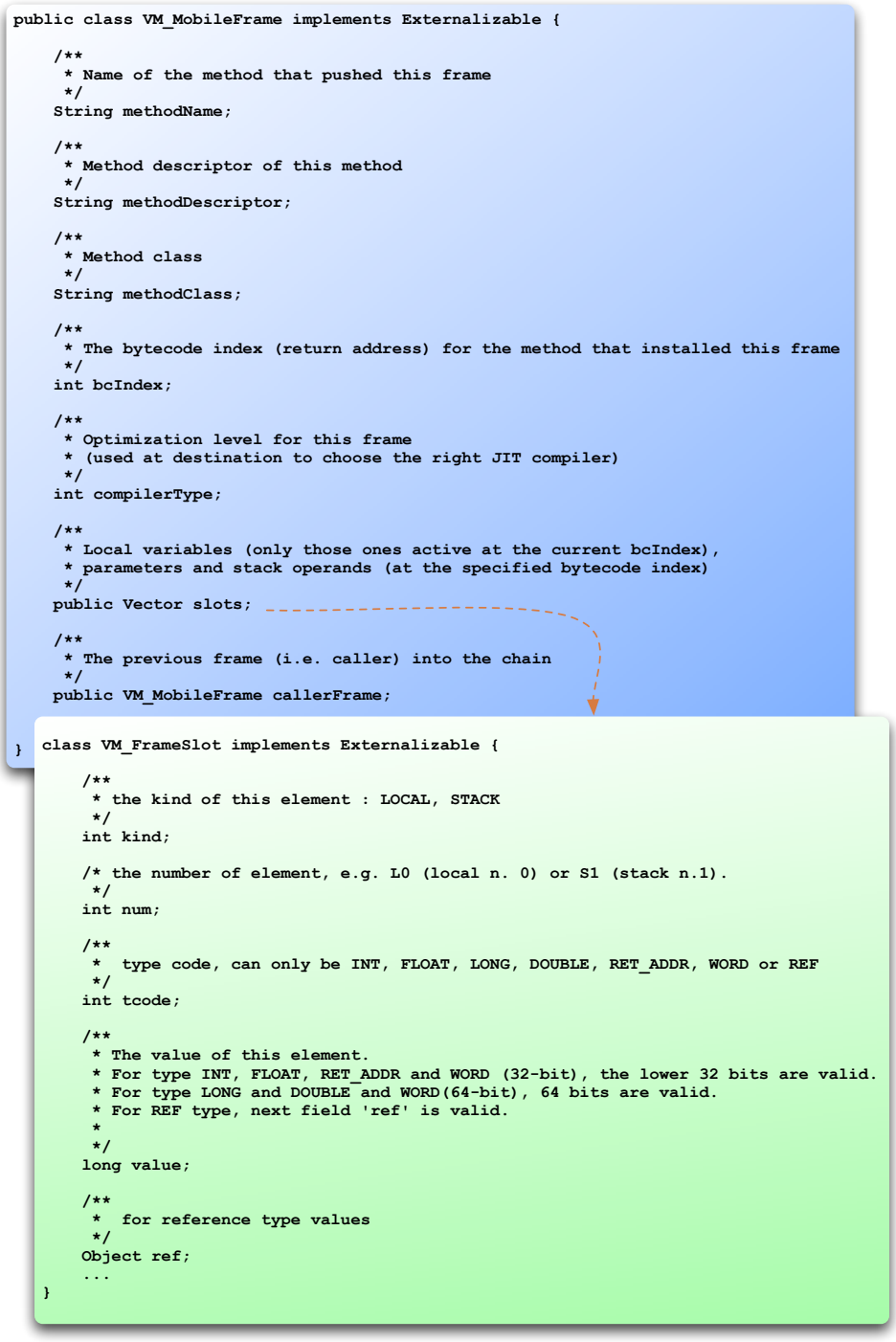


Figure 5.17: The structure of the `VM_MobileFrame` and `VM_MobileSlot` classes

The "method that pushed the frame on the stack" is the first essential information needed to restore execution at destination. Methods are class members in Java and they can be uniquely indentified by the triplet

{*fully qualified class name, method name, method descriptor*}

The first element is the full-length name of the class where the method is declared (e.g. `java.lang.String`). The sole *method name* string is not enough to precisely identify the target method, because of method overriding in Java. The *method descriptor* is the last element needed: it contains a string of characters that describe the return value and the number, types and order of the method arguments (e.g. ``()Ljava/lang/String;'' to describe the `toString()` method in `java.lang.Object`). These three elements are stored in the three `methodClass, methodName, methodDescriptor` fields of `VM_MobileFrame` in figure 5.17. For the sake of precision, it must be stated that this triplet works fine provided that the bytecode is loaded from the same codebase. Without this assumption, duplicate class names would be in conflict at classloading time.

The fourth element of the `VM_MobileFrame` is the "return address to jump to, when execution is resumed in that method". This address is retrieved at runtime from the return address stored in the callee method frame (refer to figure 4.4 or 4.5). To be fully portable on a different JVM, this address must be traslated back to a bytecode index (`bcIndex` in figure 5.17). The "bytecode index-machine code offset" mapping is maintained automatically by JikesRVM at JIT compilation time and is thus easily retrieved by Mobile JikesRVM at extraction time.

The real state of the frame is however composed by the set of local variables, method arguments and stack operands "alive" at a certain bytecode index. These are stored in the `slots` field as a `Vector` of `VM_FrameSlot` elements. Each `VM_FrameSlot` object stores information about the kind of slot (local or stack operand), the position in the Java frame, the type of variable (primitive or reference) and the value. The `value` field of `VM_FrameSlot` contains the bitwise representation of primitive types (e.g. int, float, double) and cannot be used to store object references, even though they are memory addresses in the end. The reason why a separate `ref` field has been introduced is in fact to save object reference values in a GC safe manner. If a garbage collection occurs during the state capture process, these references will be safely updated with their new address in the heap.

The set of fields in the `VM_MobileFrame` ends with two more auxiliary fields. The `callerFrame` field is naturally a pointer to the caller frame in the chain. If `null`, the chain is finished. The `compilerType` field stores the information about the kind of JikesRVM compiler that compiled the method

(baseline or optimizing). This information is exploited by the FrameInstaller to decide how to recompile the method at deserialization time.

The FrameExtractor tool

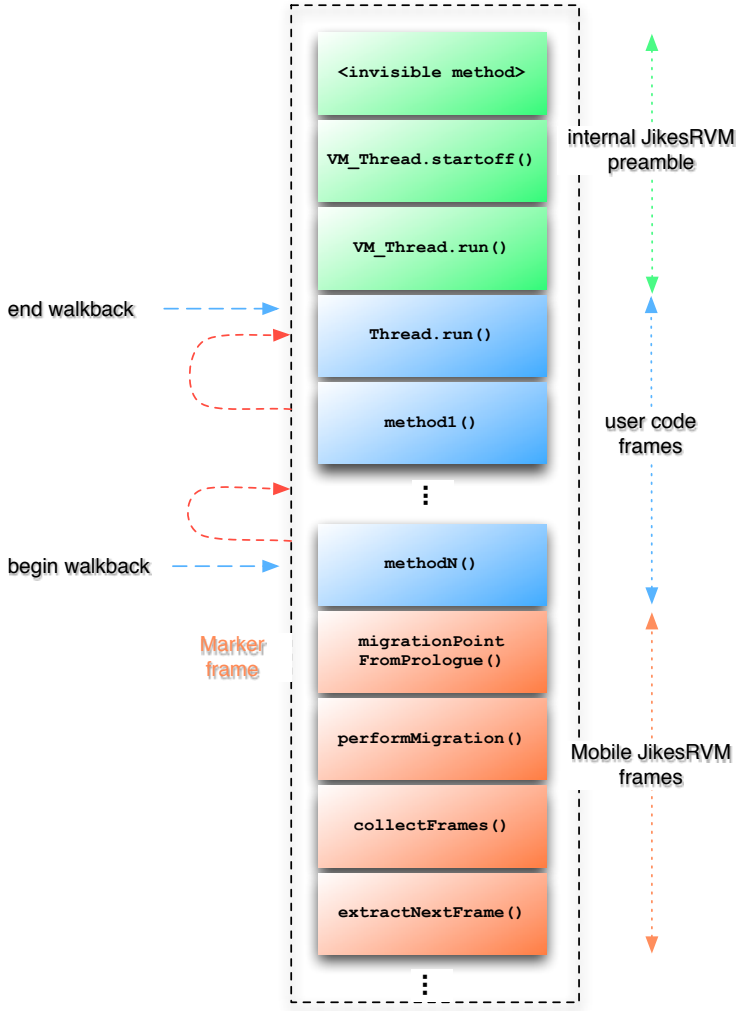


Figure 5.18: The stack walkback process

Frame extraction can be defined as the process of analyzing the call stack of the target thread and producing a set of frames containing the state of each method activation. In Mobile JikesRVM, this task is appointed to the `VM_FrameExtractor` class in the migration tools. As anticipated in the code of figure 5.14, an instance of this class is instantiated when state capture is triggered either by the thread itself (proactively) or by another thread

(reactively). In both cases, in Mobile JikesRVM the target thread is the only one responsible for self-inspecting and extracting its current state. Instead of going on with its execution, the target thread carries out the extraction process, calling the `extractNextFrame(...)` method repeatedly until the `Thread.run()` method is reached at the bottom of the stack. This process is called **stack walkback** (see figure 5.18) and in JikesRVM is typically performed when throwing an exception or when the garbage collector scans the stack to collect references for each thread.

In the case of Mobile JikesRVM, the frame extractor needs first of all

```
public VM_MobileFrame extractNextFrame() throws IOException {
    if(lastFrame)
        return null;

    /* Retrieve the current (i.e. caller) method object */
    VM_NormalMethod callerNM = (VM_NormalMethod) callerMethod.getMethod();

    /* The following code gets the machine code offset to the next instruction.
    * All operations of the stack frame are kept in GC critical section. */

    /* Get the next machine code offset of the real method */
    VM.disableGC();
        Address calleeFP =     VM_Magic.objectAsAddress(thread.stack).plus(calleeFPOffset);
        Address nextIP =     VM_Magic.getReturnAddress(calleeFP);
        Offset  ipOffset =     callerMethod.getInstructionOffset(nextIP);
    VM.enableGC();

    // Choose the right extractor for the compiler
    switch(callerMethod.getCompilerType())
    {
        case VM_CompiledMethod.BASELINE:
            currFrame=baselineExtract(callerNM, ipOffset);
            break;
        case VM_CompiledMethod.OPT:
            currFrame=optExtract(callerNM, ipOffset);
            break;
        default:
            throw new IOException("...");
     }

    // Move frame pointers up one frame
    upOneFrame();

    return currFrame;
}
```

Figure 5.19: The `extractNextFrame(...)` method of `VM_FrameExtractor`

to find the position from which to start the extraction. Walking back the stack, the extractor examines each frame to see if it is the "marker" frame for this kind of migration. In other words, the extractor searches for the frame of the `ObjectOutputStream.writeObject(...)` method, in case of a proactive migration; the marker is one of the method stubs for migration points (e.g. the `migrationPointFromPrologue(...)` in figure 5.18), for a reactive migration.

After walking back the stack up to the marker frame, the frame extractor is ready to extract the first frame from the user area of the stack. Invoking

the `extractNextFrame(...)` method of figure 5.19, the extractor examines the next frame in the stack. Depending on the kind of frame (baseline or optimizing) the extraction changes radically.

Extracting a baseline frame As observed in subsection 4.3.2, a baseline frame is always fully observable, because it keeps a regular and predictable structure. Using the bytecode maps built by the baseline compiler, the baseline extractor can retrieve the bytecode index associated to the current return address and use it to compute the types of locals and stack operands at that point. This type inference is done using the built-in `OSR_BytecodeTraverser` component. `OSR_BytecodeTraverser` does a depth first search on the bytecode array, determines the type information of locals and stack operands at a certain index. This class can only tell basic type information such as: REFERENCE, LONG, DOUBLE, FLOAT, INT, and ReturnAddress. More or less, this class does the same work as a bytecode verifier, which tells the type and size of each local and stack operand. The produced type information has to be adjusted by consulting GC maps because two different types may merge at one program point (REF and non-REF types) and to cut out reference variables that are uninitialized at the current bytecode index. Knowing the number and type of locals and stack operands, the extractor then goes directly into the frame to retrieve them at their allocated memory offset (see figure 4.4). For each extracted value, the corresponding `VM_FrameSlot` object is created and inserted into the current `VM_MobileFrame` object.

Extracting an optimizing compiled frame Frame extraction with the optimizing compiler is less immediate, because the layout of the frame is variable and physical registers can be used to store the value of locals and stack operands. As it can be seen from the code of figure 5.20, the primary information when capturing an optimized method is the *encoded osr map*, built using the mobile optimization plan of the compilation tools (subsection 5.3.1). The osr map contains a set of coordinates needed to precisely locate the various pieces of state in an optimized frame. These locations can be either spill slots in the frame spill area or physical registers. The frame extractor has therefore to reestablish the machine context that was present at the corresponding `OSR_Point`. Physical registers, saved in a special frame area before invoking the callee method, are gathered in a dedicated object of class `VM_MobileRegisters`. Now, the state is fully available partly in registers and partly in the spill area. The only thing to do is to use the osr map info to retrieve the values for each local variable and stack operand, either from a register or at a given offset in the spill area.
Another last thing that can be observed from figure 5.20 is the possibility that a single physical frame contains more than one Java frame. This

```
private VM_MobileFrame optExtract(VM_NormalMethod callerNM, Offset ipOffset) {

    byte[] stack = thread.stack;
    VM_OptCompiledMethod optCM = (VM_OptCompiledMethod) this.callerMethod;
    VM_MobileFrame frames = null;
    int regmap = 0;

    // Get osr and mc maps
    VM_OptMachineCodeMap optMCmap = optCM.getMCMap();
    OSR_EncodedOSRMap optOSRMap = optCM.getOSRMap();

    /* Retrieve the current GC map entry to check object references found in the frame */
    if(optOSRMap.hasOSRMap(ipOffset))
        regmap = optOSRMap.getRegisterMapForMCOffset(ipOffset);
    else
        throw new IOException("Opt frame: no entry in osr map for method "+optCM);

    //Retrieve machine registers from the special save area
    registers.restoreFromOptFrame(callerFPOffset, optCM);

    // return a list of states: from caller to callee
    // if migration happens in an inlined method, the state is a chain of recoverd methods.
    frames = getExecStateSequence(stack, callerNM, ipOffset,
                                  registers, optOSRMap, entry, optMCmap, regmap);

    // reverse callerState points, it becomes callee -> caller
    VM_MobileFrame prevState = null;
    VM_MobileFrame nextState = frames;

    while (nextState != null) {
        // 1. current node
        frames = nextState;

        // 1. hold the next state first
        nextState = nextState.callerFrame;

        // 2. redirect pointer
        frames.callerFrame = prevState;
        if(frames.callerFrame != null)
            frames.deinlined = true;

        // 3. move prev to current
        prevState = frames;
    }

    return frames;
}
```

Figure 5.20: The optimizating state extractor

happens, as already said in subsection 5.3.1, when the optimizing compiler performs inlining of called methods. This inlining can go on at several levels of inclusion and thus the only physical frame present will encapsulate pieces of state from all its inlinees. Once again, osr maps help the extractor figure out what state is saved for each inlinee and at what location in the physical frame. The final outcome of extracting one frame will thus be a chain of `VM_MobileFrame`s, one for each inlined method.

The FrameInstaller tool

Shown in figure 5.16, the **Frame Installer** component is the other migration tool in Mobile JikesRVM. This component is implemented by the `VM_FrameInstaller` class and is used at thread deserialization time. After reading the value of its fields from the `ObjectInputStream`, the deserialized thread receives a brand new and empty call stack. Calling the `start()` method on it, would make the thread start from the beginning, forgetting its past execution history. Such an history is written in the sequence of `VM_MobileFrame`s, serialized along with the thread object, and now restablished in memory. The Frame Installer takes essentially three actions, implemented in the code excerpt of figure 5.21:

1. trigger classloading of the method's class
2. compile self-installing versions of each method
3. install the special trampoline for the `Thread.run()` method

Loading the method class The first action consists in using the triplet {*className, methodName, methodDescriptor*} to find the method class, loading the class and returning the `VM_MobileMethod` object for that method.

Compiling self-installing method code In one of the early versions [23] of Mobile JikesRVM, frame installation was made by writing each local variable or stack operand directly in the physical frame (the frame was supposed to be only a baseline frame). This approach has been dropped in favor of self-installing specialized compilation, for two reasons:

- it heavily depends on a particular frame layout;
- it forces the thread to restart with a baseline version of the method.

The self-installing compilation technique draws inspiration from code specialization [32] in the OSR. The key insight of this mechanism is that, given the `VM_MobileFrame` of a method, we can construct a specialized method, in bytecode, that sets up the stack frame and continues execution preserving semantics. To do this, the frame installer prepends to the original bytecodes a **specialized prologue** that

```
public VM_CompiledMethod compileFrameMethods(ObjectInputStream in) {
    VM_CompiledMethod newCM = null;
    VM_MobileFrame currentFrame = chain;
    VM_NormalMethod method = null;

    currentFrame.callee_cmid=-1;

    do {
        method = currentFrame.getMethod(in);
        if (currentFrame.compilerType==VM_CompiledMethod.BASELINE)
            newCM = baselineCompile(currentFrame, method);
        else
        if (currentFrame.compilerType==VM_CompiledMethod.OPT)
            newCM = optCompile(currentFrame, method);
        else
            throw new IOException("Unknown compiler type.");

        if (currentFrame.callerFrame == null)
            break;

        currentFrame = currentFrame.callerFrame;

        // set callee_cmid of the caller
        currentFrame.callee_cmid = newCM.getId();

    } while (true);

    if(currentFrame.methodName.equals("run"))
    {
        // invoke specialRun method (which does nothing) for the first time
        if((VM_Entrypoints.specialRun.getCurrentCompiledMethod()==null) &&
           (thread != null))
            thread.specialRun();

            VM_Entrypoints.specialRun.replaceCompiledMethod(newCM);
    }

    return newCM;
}
```

Figure 5.21: Invoking the Frame Installer

- saves values into locals,
- loads values on the operand stack,
- jumps to the current bytecode index (program counter).

This mechanism allows to express the stack frame setup in the bytecode language (thus remaining immune to changes to the frame layout), and relies on the target compiler to implement the setup procedure as it sees fit (e.g. the optimizing compiler can be used to reestablish the frame instead of downgrading to the baseline each time). A graphical example of this mechanism is produced in figure 5.22.

The frame installer generates a specialized version of each inlined method, constructed such that calling the specialized root method of the inlined context restores all the inlined stack frames. The prologue of the specialized root restores the root method's state, then immediately calls a specialized version of the callee. When the specialized callee returns, the root method continues execution immediately after the call. Naturally, the procedure is applied recursively to recover from arbitrarily deep inlining.
Since the specialized method is (nearly) legal Java bytecode, a "normal" Java execution engine (interpreter or compiler) can execute the specialized method directly. Furthermore, an optimizing compiler may find more opportunities for optimization because the prologue loads runtime constant values which are not available in the original bytecode. Nevertheless, there are a few practical compilations that require modification to the target compiler. First, since the specialized version contains a prologue absent from the original bytecode, the index of each original bytecode has changed. The compiler needs to adjust bytecode indices for the specialized code in order to keep correct GC maps, exception tables, and line number tables. The OSR specialized compiler also defines a few new "pseudo-bytecode" instructions used only in the specialized prologues. Parsing these pseudo-bytecodes requires a few extra lines of code in the front-end of each compiler. One new "pseudo-bytecode" instruction is used to load literals, to avoid having to insert new constants into constant pools. Another one is a special instruction to load an address (bytecode index) to represent a return address pushed on the stack by a `jsr` instruction (bytecode jump to subroutine).

Installing the `Thread.run()` trampoline As shown in figure 5.21, each method in the frame chain is compiled with the self-installing prologue prepended to the original bytecode. Depending on the `compilerType` field of the `VM_MobileFrame` object, the frame installer uses the baseline or the optimizing compiler to produce the final machine code. However, these ad-hoc compiled methods are used only as long as that frame stays in the call stack. When it returns to the caller, new invocations of the same methods

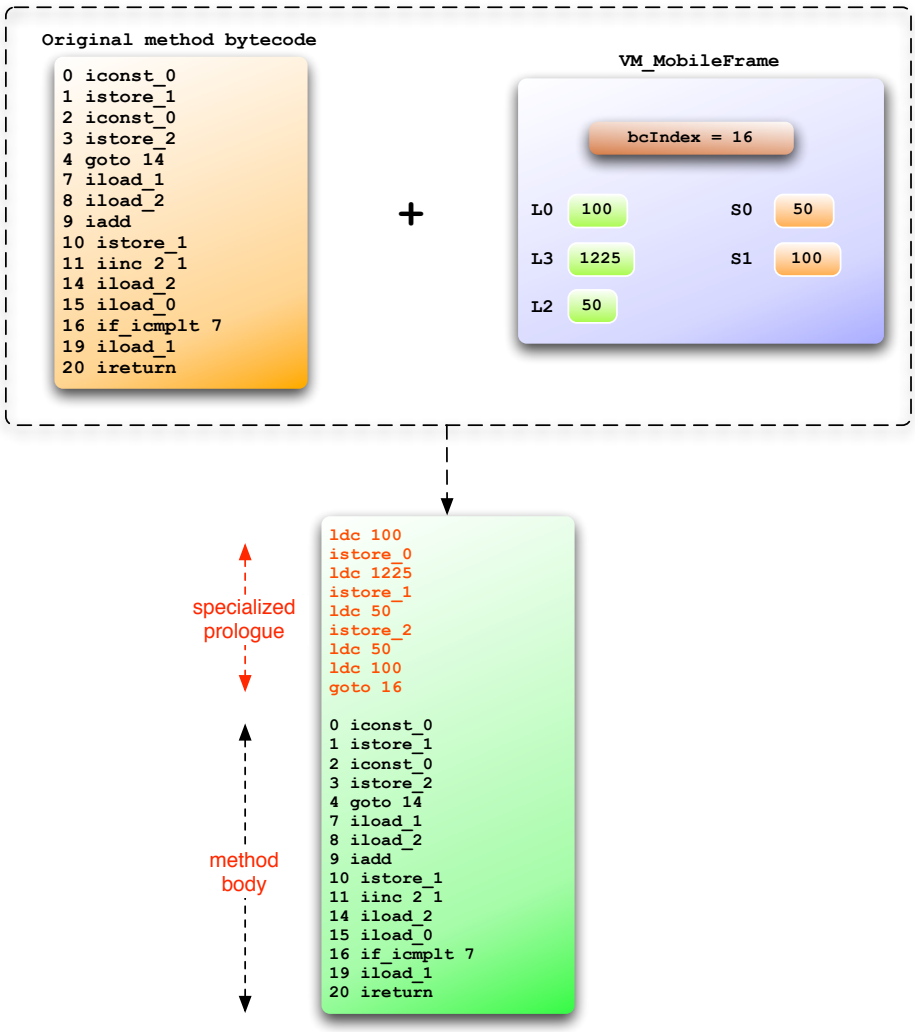


Figure 5.22: An example of a special self-installing method

will of course use the non specialized version.
At this point, the restored thread is almost ready to be started, calling `Thread.start()`. The last thing to do to make it run the chain of self-installing method bodies is hijacking the execution of the `Thread.run()` method towards the self-installing `run()` version. This is done, as shown in figure 5.21, using an auxiliary `specialRun()` method declared in the `Thread` class. `specialRun()` has an empty body and it is used as a "handle to attach" the self-installing `Thread.run()` method. When started through the `Thread.start()` method, the execution of the restored thread will thus merge into the self-installing `run()` method instead of the original one. The final outcome is that the restored thread **runs its past execution with the fast-forward and continues from the last program counter**.

5.3.3 Utility Tools

This last section is about auxiliary Mobile JikesRVM tools, useful for some kinds of applications that need a stronger control over the migration process. In subsection 5.2.1, the problem of dataspace relocation has been debated, leading to the conclusion that no unique solution can be devised for every kind of resource. In the utility tools, the experimental concept of **relocation strategy** has been implemented to provide the programmer with some fine-grained tools to deal with resources used by a mobile thread. The focus of this last paragraph is to give an idea of how the trickiest relocation policy (i.e. the "by ref") has been dealt with and implemented in Mobile JikesRVM.

The *by ref* relocation policy

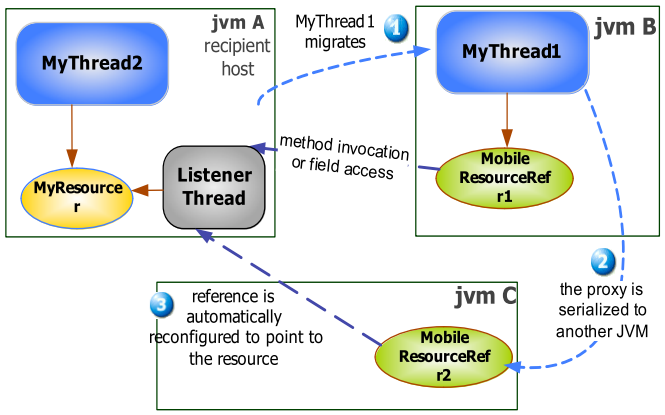


Figure 5.23: Implementation of the "by ref" relocation policy

Implementing the "by ref" policy requires hiding the details of the physical distribution as much as possible, in order to guarantee maximum transparency to the programmer: the thread can access fields and methods of the remote object, just like it would do with a local instance. Java RMI provides only a partial support to the purpose, mainly because it allows clients to only invoke methods on remote objects, while fields cannot be directly accessed (the remote object should expose proper `setXX()` and `getXX()` methods to achieve this). Moreover, RMI does not provide the needed degree of transparency, because it requires proxies and stubs to be built and compiled when a new kind of remote object is accessed. Thus, a more purpose-built protocol than RMI has been adopted, excluding features such as the "naming service" and the registry, and adding the "field access" feature and others. The resource management tool is capable of handling remote references thanks to a proper `MobileResourceRef` object (see figure 5.23), which is just a proxy to the remote object. It has exactly the same fields and methods of its remote counterpart. Furthermore, the resource management tool keeps track of the hashcode of the remote object and the hostname where it is located (called "recipient host"), so that:

- every access (through Java `getfield` and `putfield` bytecodes) to its fields triggers a communication on a socket with the recipient host, where a service thread listens to remote access requests on a predefined port;
- every invoked method is implemented as a remote invocation, served by the previous service thread.

One important difference is that the programmer should be aware of the fragility of such a reference, protecting both field accesses and method invocations with `try/catch` blocks (to deal with network failures). If a `MobileResourceRef` is either passed as a parameter of a remote method, or is set as a field value or as method return value (see step 2 of figure 5.23), the framework is able to properly handle the situation and creates another proxy on the destination (e.g. r2 on JVM B), which points to the original object on the recipient host and not to the original proxy (e.g. r1); if the proxy is passed back to the initial JVM (e.g. JVM A), the framework converts the proxy back to a reference to the real object. This behaviour avoids problems with circular remote references among proxies. Furthermore, resources with active proxies cannot be relocated by move, until each reference is released.

The resource management tool has been successfully added to Mobile JikesRVM, thanks to the aid of the special JIT compilers in the compilation tools. The extension implemented in the resource management tool practically introduces the `MobileResourceRef` concept: whenever a `MobileResource` is relocated "by ref", its binding to the migrated thread

is transparently reconfigured, so that every access to fields or methods of such resource is forwarded (through the network) to the recipient host. As for field access, the behaviour of the `getfield` and `putfield` Java language bytecodes has been extended, so that the JIT compiler can properly handle the case of a remote resource. For instance, consider the `getfield` bytecode, which should retrieve the value of a field, leaving it on top of the operand stack. The JIT compiler of Mobile JikesRVM is able to recognize `MobileResource` references, inserting a hidden method call in the compiled code whose functionality can be summarized as follows:

1. query the resource management layer to know if the current `MobileResource` object is just a proxy or a real resource object;
2. if it is a remote proxy, then send a "getfield" request to the listener thread at the destination (see 5.23) and wait until the value is ready;
3. save the retrieved value in the field of the proxy object.

Without knowing what happens beneath the surface, the migrated thread can get any fields of a resource relocated by ref, just like it would have done with a local object. There are obviously some slight differences that no layer can hide and they pertain to the usage of the network: one observation is that it is inevitable that accessing a remote field is slower than doing the same thing locally, because it implies paying the price of distribution; another crucial aspect is related to network failures which are responsibility of the programmer using a "by ref" relocated resource. Similar considerations are applicable to the `putfield` bytecode. Method invocations are instead forwarded at the recipient host, simply manipulating the proxy TIB (Type Information Block). The TIB in JikesRVM [12] is a sort of "method dispatch table" and it is referred to by an header within each instantiated object. The resource management tool modifies the TIB of the proxy object, when it is created on the destination JVM, forcing it to point to a special "remote invoker" method. The latter method simply posts an invocation request to listener thread on the recipient host and waits for the results to come back.

Chapter 6

Performance Evaluation

6.1 Introduction

Performance evaluation of a thread migration system requires taking into account every aspect of the implemented approach. From a very general standpoint, three major components should be evaluated whatever the migration system at hand:

1. the **migration cost**, comprising the time spent capturing a thread, moving its state to another JVM and restoring that state in a new thread.

2. the **overhead on the JVM runtime**, i.e. how much the thread migration system may interfere with the normal runtime execution of the JVM.

3. the **overhead on the application execution time**, which means how slower the migrable thread is executed with respect to a non migrable one. It is, in other words, the price to pay for using strong mobility in a distributed application.

In section 3.3.2, thread migration approaches have been classified in **JVM-level** and **application-level**. Depending on the chosen strategy, a certain approach may suffer or not from each of the above three overheads.

The migration cost is the most difficult to measure and it is present in all approaches, because it depends on the nature of the migrable thread and on the execution points where migration may occur. The nature of the thread means a number of factors that is unfeasible to precisely identify and scientifically measure. Factors like the code complexity of the thread or the hierarchical structure of serializable objects are all of a heuristic nature and, even if somehow measurable, they would be of little use to the programmer. For this reason, all the approaches reviewed in section 3.3 chose to limit their performance analysis to simple algorithms, like the Fibonacci recursive or factorial algorithms, taking into account only one factor contributing to the migration cost: the stack depth (number of user frames in the stack). Changing the maximum Fibonacci number to compute or the number to factorize allows, for instance, to easily tune the stack depth of the application thread, so that the trend of the migration cost can be analyzed according to the variation of the stack depth. This evaluation approach seems quite incomplete and simplistic, considering that the number of frames is just one (and perhaps one of the less important) factor impacting on the migration cost and that these algorithms are far from being representative of real distributed applications. The evaluation strategy proposed in this chapter tries to go beyond a step further than previous work, digging into the mechanisms causing migration cost in Mobile JikesRVM and evaluating them according

to more factors than just the stack depth.

Application-level migration systems are by their own nature immune from the overhead on the JVM runtime. Instead, they suffer heavily from the application execution overhead. Instrumenting application code to achieve state observability means introducing an execution overhead ranging from 88% (like in JavaGoX and Brakes) to 250% (like in JavaGo). Many of the reviewed application-level systems (e.g. Wasp [35] and STRING [48]) are accompanied by some evaluation of the overhead on the application execution time, even though with simple Fibonacci-style algorithms. On the contrary, none of the reviewed JVM-level proposals of section 3.3 seems to measure the induced overhead on the JVM itself. JVM-level approaches often introduce some kind of interferece/noise into the JVM that they manipulate and this kind of overhead is often simplisticly underestimated (e.g. claiming, like in JavaThread [18] a 0% overhead experienced running the Fibonacci recursive algorithm). JVM-level systems following the *runtime-time strategy* also introduce a non negligible overhead on the application execution time, because they perform state tracking activity during the execution flow, in preparation for the migration event.

The approach implemented in Mobile JikesRVM belongs to the migration-time subclass of JVM-level approaches. As demonstrated in section 6.3, *isolating the framework code from the rest of the JVM* and *introducing special JIT compilation only for Mobile Methods* allows Mobile JikesRVM to minimize the impact of the two overheads above to a reasonable and sometimes even negligible cost.

Aware that no benchmarks are available for thread migration systems, the strategy followed in this dissertation work has been that of "adapting some well-known JVM benchmarks" in order to

- measure the JVM-runtime and the application execution overhead, in a realistic and repeatable way;
- analyze how the migration cost behaves in the presence of a comprehensive and diverse set of application code.

The chosen benchmark suite is briefly described in the next section.

6.2 DaCapo Benchmarks

Dacapo describes [4] itself as

> *"This benchmark suite is intended as a tool for Java benchmarking by the programming language, memory management and computer architecture*

communities. It consists of a set of open source, real world applications with non-trivial memory loads. The suite is the culmination of over five years work at eight institutions, as part of the DaCapo research project, which was funded by a National Science Foundation ITR Grant."

Industry and academia typically use the **SPEC** Java benchmarks (i.e. the SPECjvm98 and the SPECjbb2000 [66, 67]) to evaluate the performance and integrity of a JVM implementation. SPECjvm98 and SPECjbb2000 are proprietary and closed benchmarks, while the DaCapo suite is a set of general-purpose and freely available Java applications. With respect to SPEC, DaCapo is more complex in terms of static and dynamic metrics. For example, they have much richer code complexity, class structures, and class hierarchies than SPEC. This complexity produces a wider variety and more complex object behavior at runtime, as measured by data structure complexity and heap composition.
DaCapo provides the JVM designer with diverse programs that maximize coverage of application domains and behaviors. They focus on client-side, non GUI applications that are easier to measure in a completely standard way. The benchmark suite is packaged as a single jar file containing a harness, eleven benchmarks, the libraries they require, three input sizes (i.e. *small, medium, large*) and input data. The eleven benchmarks are the following:

1. **antlr**, a parser generator and translator generator
2. **bloat**, a bytecode-level optimization and analysis tool for Java
3. **chart**, a graph plotting toolkit and pdf renderer
4. **eclipse**, an integrated development environment (IDE)
5. **fop**, an output-independent print formatter
6. **hsqldb**, an SQL relational database engine written in Java.
7. **jython**, a python interpreter written in Java
8. **luindex**, a text indexing tool
9. **lusearch**, a text search tool
10. **pmd**, a source code analyzer for Java
11. **xalan**, an XSLT processor for transforming XML documents

The DaCapo harness is used to invoke the benchmarks and perform a validity check that ensures that each benchmark ran to completion correctly. The validity check performs checksums on `System.err` and `System.out` streams

during benchmark execution and on any generated files after benchmark execution. The harness passes the benchmark if its checksums match precalculated values.
To find a performance-stable iteration, the harness takes a window size omega (number of executions) and a convergence target `nu`, and computes the standard deviation `sigma` and the arithmetic mean `mu` of the last `omega` execution times. It runs the benchmark repeatedly until either the coefficient of variation `sigma/mu` of the last `omega` runs drops below `nu`, or reports failure if the number of runs exceeds a maximum `m`. Once performance stabilizes, the harness reports the execution times of the next iteration.

6.3 The chosen evaluation strategy

The DaCapo benchmarks have been used to evaluate Mobile JikesRVM in a realistic and meaningful way. The good news with DaCapo benchmarks is that they have been developed by some of the researchers in the JikesRVM project and they are thus fully supported by JikesRVM. Nonetheless, it must be pointed out that JikesRVM has been successfully tested also with other benchmark suites, like the SPECjvm98 and SPECjbb2000.

The test environment used to carry out the measurements is the following:

- Pentium IV, 3.4Ghz, 1GB RAM
- Ubuntu Linux 7.04
- JikesRVM version 2.4.6, "production" configuration, i.e. Fast Adaptive with assertion checking disabled and a copying generational garbage collector with a non-copying mark-and-sweep mature space.
- GNU classpath release 0.92
- DaCapo benchmark suite release 10-2006, with the "converge" option enabled.

6.3.1 Overhead on the JVM runtime

As stated above, JVM-level approaches should take into serious consideration the **implicit overhead they may introduce in the JVM runtime**. Even though some reviewed works (e.g. in **JavaThread** [18]) claim to introduce a negligible overhead on the JVM they manipulated, this statement seems never supported by a thorough evaluation with some full-fledged benchmarks. Moreover, being able to run a complete set of benchmark applications on the modified JVM is a further proof that the modifications did not tamper with the integrity of the JVM itself. In the present research work,

Overhead on the JVM

Benchmark name	JikesRVM (sec)	Mobile JikesRVM (sec)	Overhead (%)
antlr	193.40	194.50	0.57%
bloat	1230.20	1249.80	1.57%
chart	2019.00	2058.40	1.91%
eclipse	14026.00	14503.00	3.29%
fop	587.60	604.00	2.72%
hsqldb	464.20	468.50	0.92%
jython	1041.00	1059.00	1.70%
lusearch	2399.60	2416.00	0.68%
luindex	501.20	511.80	2.07%
pmd	58.60	58.80	0.34%
xalan	22647.00	22899.00	1.10%

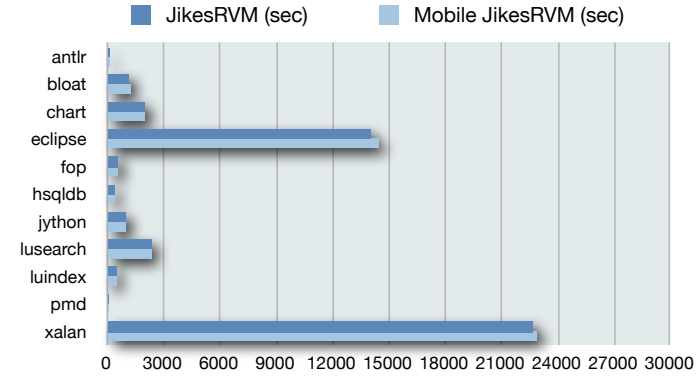


Figure 6.1: Measuring the overhead on the JikesRVM

the overhead introduced by Mobile JikesRVM on the JikesRVM has been measured running all the eleven DaCapo benchmarks, as reported in figure 6.1. The table shows the average times reported by the DaCapo harness for the eleven benchmark applications in the suite. The time is computed using the performance-stable methodology described in section 6.2, with a window size `w = 10`, a maximum number of iterations `m = 20` and a target coefficient of variation `sigma/mu = 3.0`. The leftmost column reports the execution times on an untouched version of the JikesRVM, while the central column shows the times measured with the Mobile JikesRVM installed. The rightmost column computes the percentage overhead introduced by Mobile JikesRVM, which appears very limited (no more than 3.29% for the *eclipse* benchmark) and often even negligible (i.e. under 1.00%). The above times give an idea of "how much Mobile JikesRVM may interfere with the normal runtime execution of JikesRVM". This little overhead can be basically imputed to two factors:

1. the mechanism to determine whether a loaded method is to be a Mobile Method or a regular one. This piece of code has been reported and explained in figure 5.6 of subsection 5.3.1.

2. the test introduced at JIT compilation to decide what JIT compiler to use (i.e. one from the Compilation Tools or from the standard JikesRVM set). The code that implements this test has been shown in figure 5.7 of subsection 5.3.1.

As a rule of thumb, it can be stated that in Mobile JikesRVM *applications with more methods to load dynamically and to JIT compile (like the eclipse benchmark) are more affected by the overhead on the JVM runtime.*

6.3.2 Overhead on the application execution time

The times in the previous subsection have been collected without using the "migration-friendly" compilers from the **Compilation Tools**. In order to measure the **overhead on the application execution time**, each method body in the DaCapo benchmark suite must be loaded as a **MobileMethod** and compiled by the special compilers (baseline and optimizing) presented in subsection 5.3.1. The effort to make this possible has been nearly zero, thanks to the MobileMethod concept and to the `RVM_MOBILECODE` environment variable (see section 5.3.1). The code of the DaCapo benchmarks did not need any modification with keywords or markers to be considered "mobile". Simply setting the `RVM_MOBILECODE` variable to point to the DaCapo jar file

```
export RVM_MOBILECODE=/home/rquitadamo/dacapo-2006-10.jar
```

enabled the special JIT compilation for MobileMethods. The table in figure 6.2 shows the execution times of the DaCapo benchmarks when compiled

Performance slowdown of Mobile Methods

Benchmark name	*Without Mobile Method compilation (sec)*	*With Mobile Method compilation (sec)*	*Overhead (%)*
antlr	*194.50*	*202.00*	*3.71%*
bloat	*1249.80*	*1294.00*	*3.42%*
chart	*2058.40*	*2150.00*	*4.26%*
eclipse	*14503.00*	*15910.00*	*8.84%*
fop	*604.00*	*641.00*	*5.77%*
hsqldb	*468.50*	*487.00*	*3.80%*
jython	*1059.00*	*1096.00*	*3.38%*
lusearch	*2416.00*	*2502.00*	*3.44%*
luindex	*511.80*	*538.00*	*4.87%*
pmd	*58.80*	*63.00*	*6.67%*
xalan	*22899.00*	*24002.00*	*4.60%*

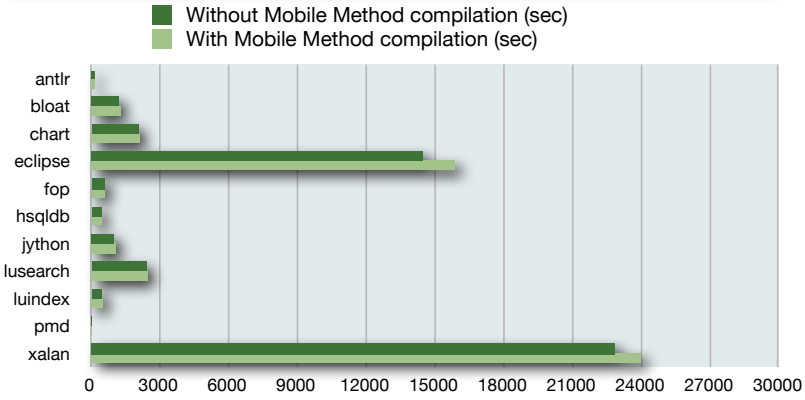


Figure 6.2: Performance slowdown of Mobile Methods

either by the standard JikesRVM compilers (leftmost column) or by the Compilation Tools of Mobile JikesRVM (central column). The percentage overhead is calculated in the righmost column and ranges from 3.42% to 8.84%.
As easily predictable, the execution time of a Mobile Method cannot be the same as a regular method. As discussed in section 3.2, for an execution state to be captured, the JIT compiler may be asked to limit its potentials in favour of a full state observability and a smoother granularity of interruption. In the light of the latter considerations, the computed overhead in figure 6.2 can be easily motivated as concerns Mobile JikesRVM:

1. a Mobile Method (both baseline and optimizing) is compiled using Migration Points instead of yieldpoints (see subsection 5.3.2). This means that when a migration point is taken by the thread, a couple of additional test is needed to figure out if a scheduling yield or a reactive migration has been requested on that thread. These tests have been described in figure 5.11 of subsection 5.3.2. In order to reduce as much as possible the impact of these tests, the method stub of a Migration Point is precompiled optimized in JikesRVM bootimage.

2. an optimized Mobile Method is compiled inserting OSR maps at method calls. These maps have been introduced in subsection 5.3.1, highlighting their importance when coping with aggressive code optimizations by the JIT compiler. In that same subsection, the effect of an `OSR_Point` IR instruction has been pointed out. Since the compiler considers all method variables (locals and stack operands) live at an `OSR_Point`, it artificially extends the live ranges of variables and limits the applicability of optimizations such as dead code elimination, load/store elimination, alias analysis, and copy/constant propagation.

Nevertheless, it turns out from the times of figure 6.2 that the above "constraining aspects" have often a very limited impact on the performance of the benchmark applications and thus represent a more than fair tradeoff for many kinds of distributed and parallel applications. Yet, these numbers are far from the ones of many code instrumenting approaches in the literature (e.g. 88% up to 250% for JavaGo [62], JavaGoX [61] and Brakes [79]). Other JVM-level systems (e.g. JavaThread [18] or CIA [41]) claim, of course, that their overhead is significantly lower than these overwhelming times (e.g. 0% in the case of JavaThread). However, they support these assertions using simplistic recursive applications (e.g. the Fibonacci algorithm), with trivial code complexity. For instance, running the Fibonacci algorithm with Mobile JikesRVM the measured overhead has been always under 0.5%, but this result does not give a sound idea of what is the real price to pay for having full observability and migration. The code complexity and workload of these simple recursive programs is too small to manifest a perceptible difference

in the execution time when compiled as Mobile Methods. This is the reason why the DaCapo benchmarks have been adopted to measure this overhead as well.

6.3.3 Analyzing the migration cost

The third, and certainly the most used, metric to evaluate the performance of a thread migration system is the **migration cost**. This cost can be measured only when a thread migration is initiated either by the thread itself (proactively) or by another thread (reactively). Despite of the way migration is triggered, five subsequent phases contribute to the overall migration cost:

1. the **capture time**, i.e. the time spent extracting from the physical thread stack a sequence of logical frames. Together with the `Thread` object and all its referenced objects, the frame chain forms the extracted state of the target thread.

2. the **serialization time** is the time needed to convert the Java objects captured in the previous phase into a binary portable representation (i.e. using the Java Object Serialization protocol).

3. the **transfer time** is the physical latency needed to move the binary thread state from one host to another in the network. It does not occur when the state is simply checkpointed to a local disk.

4. the **deserialization time** is spent translating the binary thread state back into a set of Java objects on the target JVM.

5. the **restore time**, i.e. the time to reestablish the call stack of the migrated thread and resume its execution on the new JVM.

The problem of giving the programmer a meaningful evaluation of such times is by all accounts a non trivial one. The reasons are remarkably straightforward:

1. *they heavily depend on the particular application*

2. *they can change significantly depending on when the migration occurs*

On the on hand, the first point says basically that it would be impossible to precisely predict the migration cost an application is going to experience, because that cost depends on factors for whom no unique metric exists. It is, for instance, very difficult to predict the time needed to serialize a Java object: the serialization time depends on the hierarchical depth of the object class, on the number of fields, their type (e.g. reference or primitive) and their value (e.g. a reference field with a null value is serialized much faster). These aspects of serialization along with its recursive nature (objects pointing to other objects in a tree style) make it extremely hard to

precisely define some metric to estimate the cost of serializing an object. On the other hand, the second point adds the temporal dimension to the problem, bringing further complexity. In other words, the migration cost varies according to the current execution point where the target thread is captured. This consideration motivates in the literature the criticized choice of using recursive algorithms to prove that a migration framework behaves well, simply because the migration cost grows proportional to the number of frames in the call stack. Such an approach clearly manifests the weakness of using one single metric (the stack depth) to evaluate a time that strongly depends on other factors as well.

In the remainder of this section, a deeper analysis of the capture and restore times is proposed, using the DaCapo benchmark suite to realistically measure them in a repeatable way. The major factors contributing to the capture and restore times in Mobile JikesRVM are identified and each factor is analytically correlated with the corresponding time. The other three times are instead not considered because

- they are neither caused nor influenced by Mobile JikesRVM (the serialization and the socket IO performance depends on JikesRVM and on the GNU classpath implementation of the Sun JDK),
- the applications in the benchmark suite are not supposed to be serialized and it would be unfeasible to make them as such.

Nonetheless, an emphirical measure of their impact will be given in chapter 7, where the attention will focus on some real applications using strong thread migration as an architectural asset.

Capture and Restore Times in Mobile JikesRVM: the experiment

Subsection 5.3.2 outlined the mechanism used to implement the two migration tools: Frame Extractor and Frame Installer. Starting from that knowledge, the main factors affecting the capture and restore times can be isolated:

- **number of frames**: the iterative process of state capture or restoration is repeated identical for each frame in the call stack, so that the overall capture (or restore) time is the sum of `n` per-frame capture (restore) times. This consideration suggests that both the times will grow more or less proportional to the number `n`. The "more or less" becomes "certainly" in case of identical frames (e.g Fibonacci recursive algorithm).
- **frame size**: it is intended as the number of method arguments, local variables and stack operands alive in the method frame (hereafter

called only *slots*). This number influences the per-frame capture and restore times, because the state of the frame is bigger and it takes longer to capture or restore it.

- **bytecode size**. The length of the method body may influence the capture time and, especially, the restore time: the Frame Installer creates a self-installing method version (bytecode) for each frame and compiles it to machine code. The latter compilation time depends clearly on the size of the method bytecode.

- **compiler type**: baseline capture is quite different from optimizing capture, because it uses different techniques to track and extract the state. Frame restoration changes only for the type of compiler used to recompile the self-installing method.

These identified factors are easily measurable during the frame extraction and restoration process. For this reason, both the Frame Extractor and the Frame Installer have been equipped with a boolean `benchmarking` flag (default to `false`). When the `RVM_BENCHMARKING` boolean variable is set to `true` before starting the benchmark harness, every time a frame is captured (or restored) a benchmarking record is created with the data about

1. the current frame size (in number of slots),

2. the bytecode size of the "pushing method" (in bytes),

3. the compiler type (optimizing or baseline).

Each record is then filled in with the measured capture or restore times. At the end of the capture (or restore) process these records are appended to a log file as lines of space-separated fields (to be processed and plotted as shown later).
A temporary trick has been exploited to trigger thread migration during the execution of the DaCapo benchmarks, without having to modify the benchmark code. Only for the duration of the experiments, each yieldpoint has been treated as a migration point (keeping the `takeMigrationPoint` flag of figure 5.11 to `true`). The final outcome is that the threads in the DaCapo benchmark will undergo a "fake migration" every time a yieldpoint is taken (i.e. the timeslice expires). The migration is willingly "fake" because the frame chain extracted is restored straight away, for the sole purpose of measuring the capture and restore times (no serialization or deserialization is performed, no state transfer occurs and the restored thread is never started). After each "fake migration" the benchmark thread continues normally, just as it would have done with a regular yieldpoint.

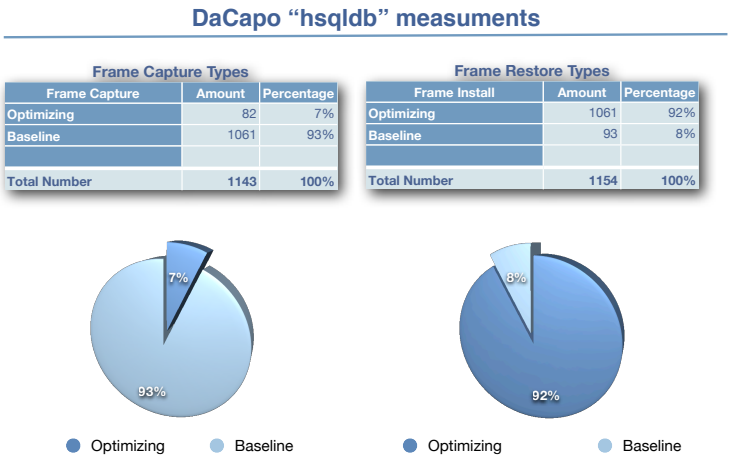


Figure 6.3: Distribution of baseline and optimizing frames in the hsqldb experiment

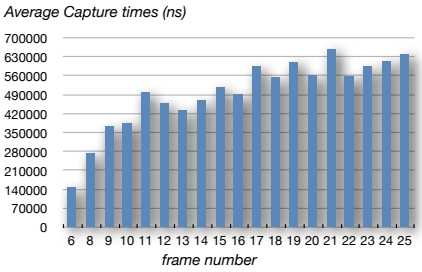
These special benchmark runs produce performance data in the form of two log files (`captureTimes.log` and `restoreTimes.log`) for each benchmark in the DaCapo suite. In the following subsection, the times of only one benchmark application (the “hsqldb”) are analyzed and discussed, for the sake of brevity and because the same behaviors have been experienced with all the other ten benchmarks of the DaCapo suite. The reason for choosing **hsqldb** was that it exhibited the highest number of optimizing frames at capture time, thus allowing a more precise analysis of both kind of frames. The ratio of baseline and optimizing frames for **hsqldb** has been reported in figure 6.3.

Analyzing the `hsqldb` benchmark

Figure 6.4 and 6.5 show the trend of the overall capture and restore time for both baseline and optimizing frames. The first result is that the capture time (both baseline and optimizing) stays lower than 1ms, even for a stack depth of 25 frames. Such a time grows logarithmic with the number of frames rather than proportial; this can be explained considering that, in a non artificial situation, different methods may push their frames on the stack, with changing frame sizes and bytecode lengths. Further noise in the data is introduced by the **garbage collector** that is triggered in complex long-running applications more than in simple short-running code. Garbage collection will be, in fact, source of sporadic peaks in the all the measured

Overall Baseline Capture Times

Number of frames	Average times (ns)
6	150000.00
8	276000.00
9	374000.00
10	389571.43
11	501500.00
12	462666.67
13	437500.00
14	472333.33
15	519000.00
16	493000.00
17	594253.73
18	557700.00
19	613053.57
20	565320.51
21	665451.61
22	561577.98
23	594871.79
24	618000.00
25	645887.32



Overall Baseline Restore Times

Number of frames	Average times (ns)
6	1261000.00
8	2453000.00
9	2527000.00
10	8652857.14
11	4327250.00
12	10112666.67
13	12334500.00
14	16045666.67
15	14607750.00
16	20403000.00
17	14978750.00
18	12961750.00
19	15646666.67
20	22339750.00
21	22182000.00
22	31835200.00
23	34045416.67
24	25961000.00
25	29454000.00

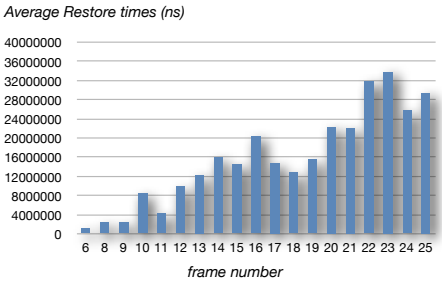
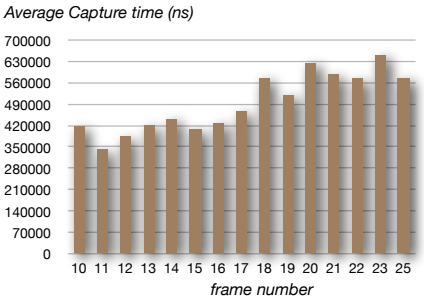


Figure 6.4: The overall capture and restore times trend for baseline frames

times and cannot be ignored or suppressed.
Frame installation is instead more expensive (but still in the order of milliseconds), because it includes the bytecode recompilation time. Its trend is more proportional than the capture time, due to the predominance of the recompilation time (which grows proportional as it will be clear). It may be noted that the frame numbers reported on the x-axis may present gaps because in the used benchmarks there is no way to manually increment the stack depth, like in a recursive algorithm.

Overall Optimizing Capture Times

Number of frames	Average times (ns)
10	418000.00
11	343000.00
12	386000.00
13	424000.00
14	445000.00
15	411000.00
16	431000.00
17	469400.00
18	574333.33
19	523333.33
20	626222.22
21	588285.71
22	577166.67
23	654600.00
25	577000.00



Overall Optimizing Restore Times

Number of frames	Average times (ns)
10	8577500.00
11	8275000.00
12	15811000.00
13	11460900.00
14	18467000.00
15	11354000.00
16	11403000.00
17	23665833.33
18	17118000.00
19	15814250.00
20	21771181.82
21	22858875.00
22	30397307.69
23	31437478.26
25	37454000.00

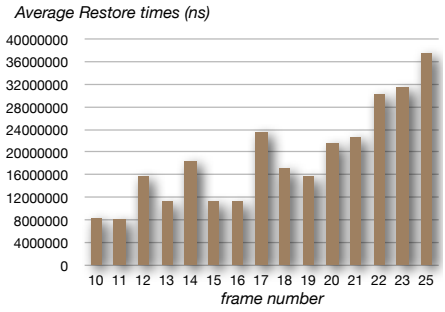


Figure 6.5: The overall capture and restore times trend for optimizing frames

Decomposing the capture time The overall capture time measured earlier is now decomposed in its major parts to investigate further where the identified factors play their role. The three most "computationally intensive" activities in baseline frame capture are:

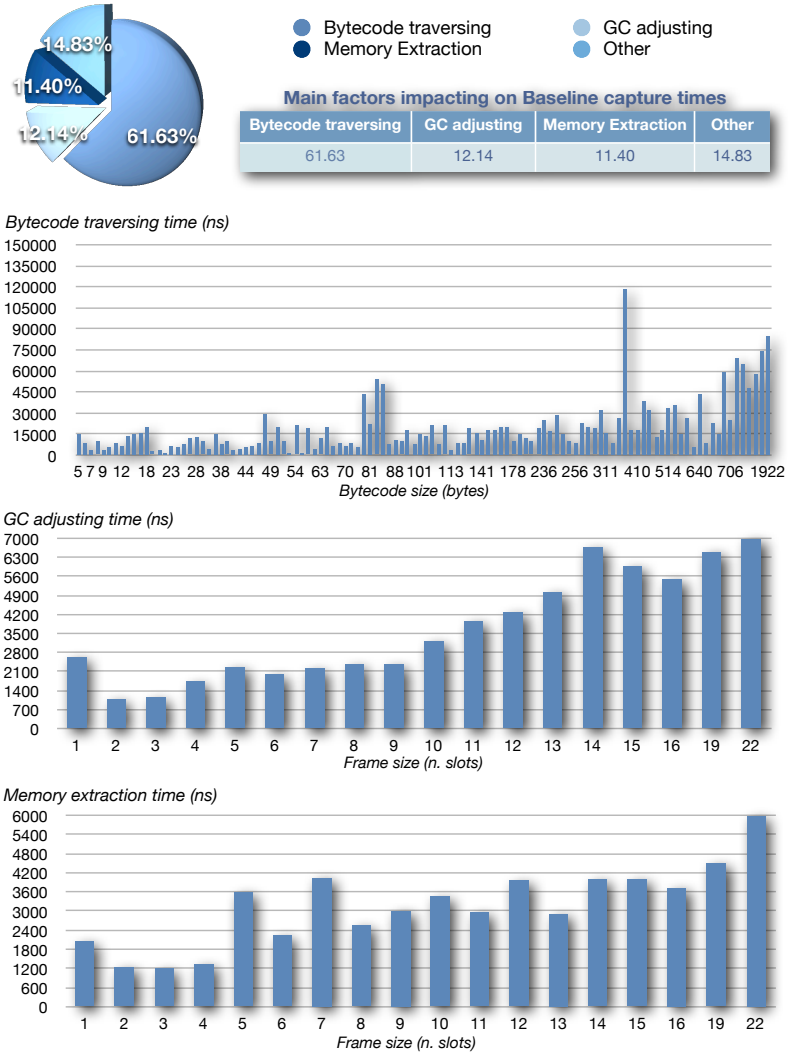


Figure 6.6: The major components of the baseline capture time

1. **bytecode traversing** with the `OSR_BytecodeTraverser` (see subsection 5.3.2). Bytecode traversing means reading the bytecode with a depth-first search strategy and it is used only in the baseline frame extractor to build the set of locals and stack operands (slots) alive at a certain bytecode index, along with their type information.

2. **Garbage Collector (GC) types adjusting**: the extractor queries the GC maps (built by the baseline compiler) to cut out reference slots whose value is considered uninitialized at the current bytecode index.

3. **Memory extraction from the physical frame** is the actual frame extraction. It reads from the physical frame the current value of each slot and adds the corresponding element into the `VM_MobileFrame`.

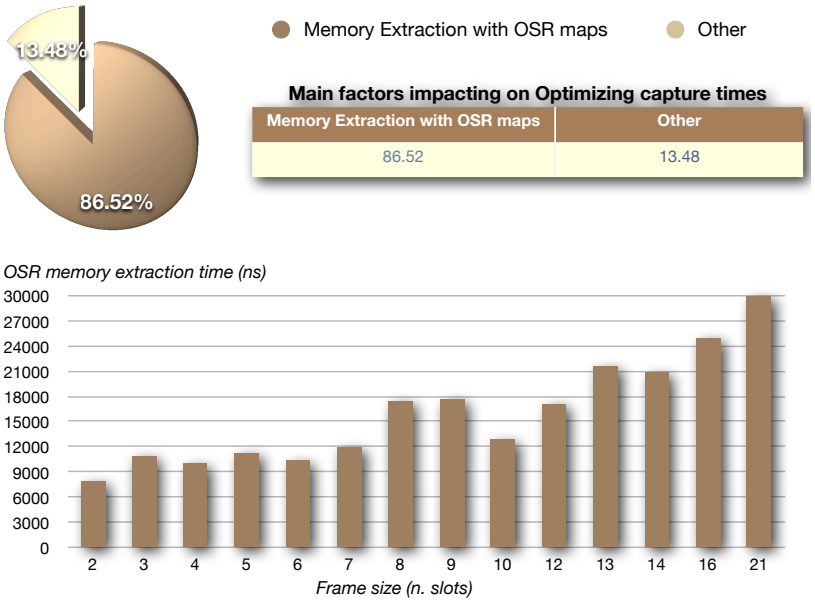


Figure 6.7: The major components of the optimizing capture time

The average impact of these phases on the baseline capture time has been calculated and shown in the pie chart of figure 6.6. It comes out that the bulk of the capture time for a baseline frame is spent in the bytecode traversing activity. This suggests that the overall capture time can be further optimized in the future, if bytecode traversing is dropped and replaced by some other mechanism, similar to the OSR maps in the optimizing frame capture. Figure 6.6 also shows how the three activities above vary according to two factors: bytecode length and frame size. The former influences the bytecode traversing time and the approximate trend in the experiment seems to more

exponential than proportional. This is another important argument in favor of a future replacement of this type inference mechanism. As concerns the other two activities (GC adjusting and memory extraction), their measured times grow gently and proportional to the frame size (number of slots).
Optimizing capture is instead less complex than the baseline. From figure 6.7, it can be observed that the only major activity performed is **memory extraction with the OSR map**. The availability of the OSR map is a clear advantage with respect to the baseline capture, because the number and types of the slots is a priori encoded and ready to be used.

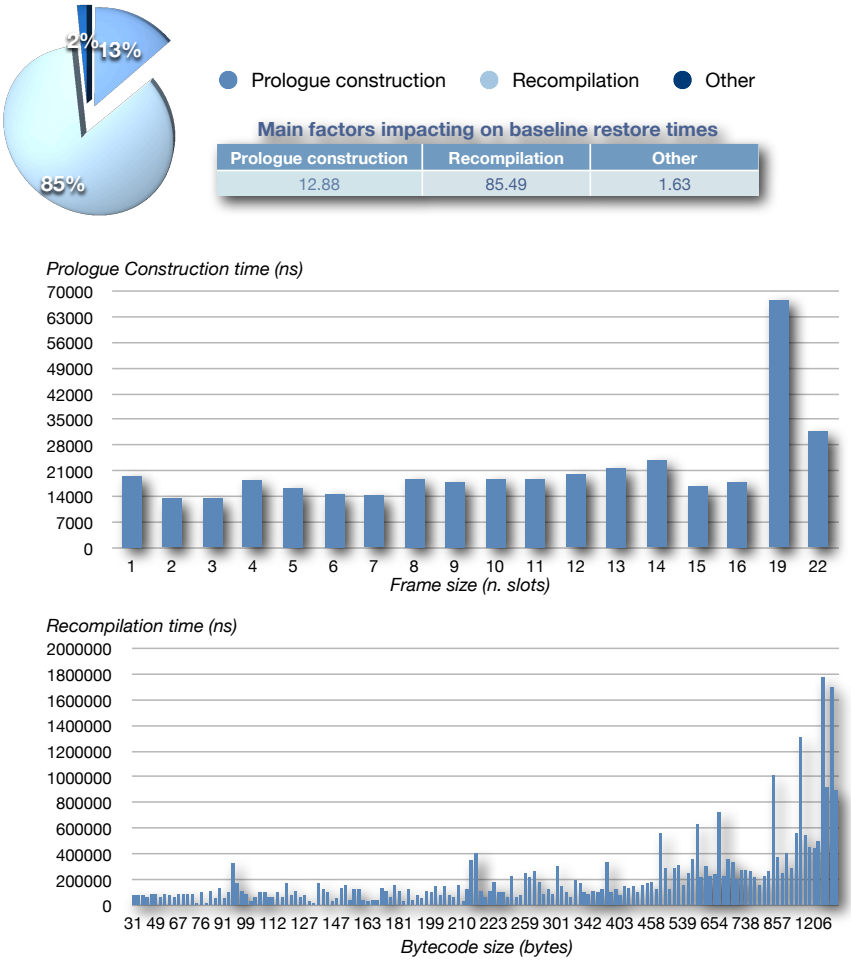


Figure 6.8: The major components of the baseline restore time

Decomposing the restore time Frame restoration in both the baseline and the optimizing installer comprises two activities:

1. **bulding the self-installing prologue**. A set of specialized instruc-

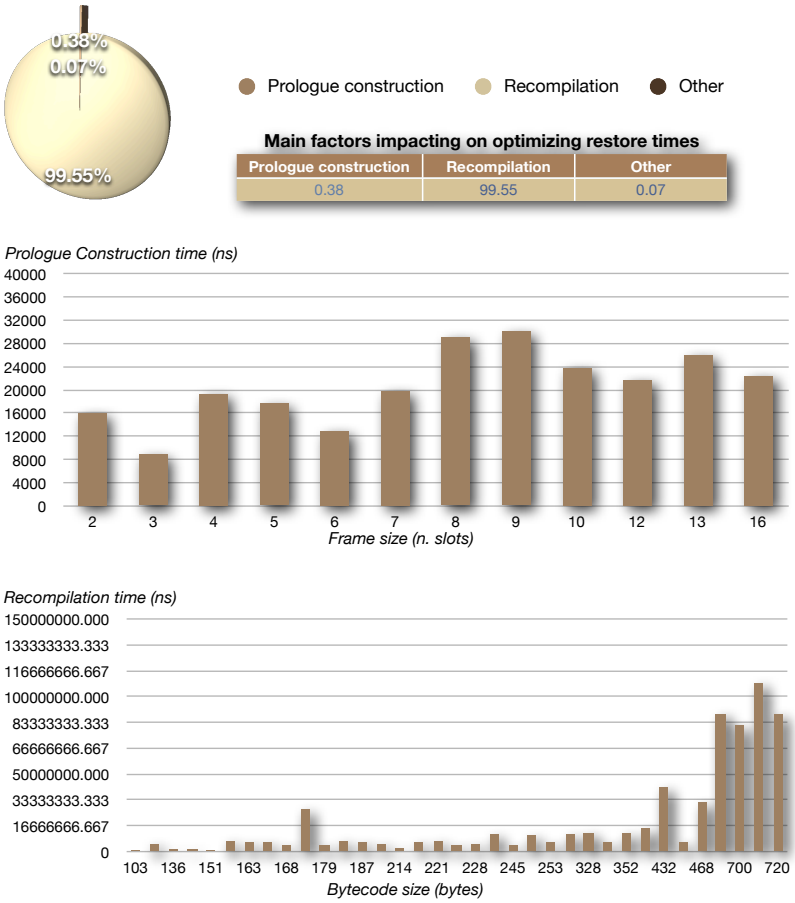


Figure 6.9: The major components of the optimizing restore time

tions is built to restore the value of each frame slot at the beginning of the method (see section 5.3.2).

2. **recompiling the self-installing method with the corresponding compiler**. The self-installing method with its sythentic prologue must be recompiled with the optimizing or baseline compiler, depending on the `compilerType` field of the `VM_MobileFrame` (the compiler type used on the source JVM when the frame was captured).

Figures 6.8 and 6.9 show the mean percentage impact of these two activities on the overall restore time.
As it is predictable, optimizing compilation is much more expensive than baseline compilation. This is the reason why the impact of the recompilation activity is heavier in the optimizing restore time than in the baseline (99.5% versus 85%). The prologue construction activity grows very slowly with respect to the frame size (it is not dependent on the bytecode size or on the compiler type). The recompilation time grows instead almost exponential to the bytecode length for both the baseline and the optimizing compiler. This behavior do not however depend on Mobile JikesRVM and cannot be changed or optimized.

6.4 Final remarks

The performance analysis proposed in this chapter leads to some final remarks:

1. Every JVM-level migration system should carefully test the modified JVM with realistic and complex benchmarks, in order to discover (and fix) possible unforeseen performance slowdown and security bugs introduced. Mobile JikesRVM has been compared with a pristine JikesRVM and the mean measured overhead has been encouragingly low (always under 3% and very often under 1%).

2. Full state observability and thread interruption do not come for free. Their cost has been measured in a repeatable way, treating the eleven DaCapo benchmarks as Mobile Methods. The insertion of migration points and OSR maps (for optimizing methods) causes a very limited and acceptable slowdown in the performance of the benchmark (it ranges from 3.42% to 8.84%).

3. The cost of thread migration is extremely hard to measure in a meaningful way. It depends on the particular application and there is no well-assessed set of metrics to evaluate it. In this work, the focus has been on the capture and restore times, which are the only two times strictly related to Mobile JikesRVM (serialization, transfer and deserialization are too application-dependent to be meaningful measured).

4. Capture and restore times have been decomposed into their major activities and a set of factors (i.e. frame number, frame size, bytecode length and compiler type) influencing those activities has been identified. Each activity has been sampled (using the DaCapo benchmark instead of the canonical and simplistic recursive Fibonacci algorithm) and its times plotted. From the measured times, it emerges that thread capture in Mobile JikesRVM is very fast (less than 1ms even with 25 frames in the stack), with each activity growing almost always proportional with respect to the responsible factor. Frame restore is instead more expensive due to the impact of the recompilation time, but still under the 40 msecs for a large number of frames.

This analysis has been intended to give a hint of how much the techniques and the approach used in Mobile JikesRVM cost in a realistic set of Java applications. It does not pretend to give the programmer a precise measure of how strong thread migration will perform in her distributed applications, when using Mobile JikesRVM. Aware that too much depends on the particular application, the position of this thesis is that *trying to identify a comprehensive set of metrics to estimate the migration cost is an unfeasible effort and the bigger the number of metrics, the less useful they are to the programmer*. Chapter 7 takes two real-life applications and shows how Mobile JikesRVM has been exploited and what has been the migration cost experienced in those cases.

Chapter 7

Application Scenarios of Mobile JikesRVM

This last chapter reports on two real application scenarios, where the present strong mobility approach implemented in Mobile JikesRVM has been successfully exploited and tested. Such applications are the product of a 6 months research visit, spent by the student during his last PhD year, at the Institute for Human and Machine Cognition (IHMC) [5], in Pensacola, Florida, US. Working in collaboration with the NOMADS team [74] and especially with prof. Niranjan Suri, strong thread migration has been concretely applied to two research projects the **PIM** and **Agile Computing** - funded respectively by NASA together with the US Office of Naval Research and by the USA AFRL (Air Force Research Lab). As it will be explained in the following two sections, strong mobility in Java has been of paramount importance for both the projects, because it is the "architectural keystone" to achieve features like:

- simple and efficient robot coordination;
- fault-tolerance and robustness in distributed applications;
- computational load distribution in mobile ad-hoc service-oriented architectures;
- opportunistic exploitation of transient and mobile resources.

It must be pointed out however that the work presented here is ongoing and future research and the results obtained are therefore still preliminary and exploratory.

7.1 Robot Coordination with the PIM

Many scenarios ranging from search and rescue to combat operations can benefit from teams of humans, robots and computers that collaborate and coordinate together to solve a problem. Achieving coordinated behaviour among multiple physical or logical entities is a challenging problem, especially in uncertain and possibly hostile environments.
Current research in this field has produced so far two categories of approaches to coordination:

1. the **centralized approach**, with a single coordinating authority that directs and coordinates the activities of all team members. The coordinating authority needs to have complete and up-to-date information about the operational state of each of the robots.
2. the **distributed (negotiation-based) approach** where robots are controlled by software agents often organized in a Multi-Agent System (MAS). Each robot is responsible for its own actions and maintains its own world-view. Coordination amongst the agents requires something

akin to social negotiation or, like in biologically-inspired approaches, to an emergent behavior mediated by the environment (*stigmergy* [40]).

The centralized approach, simple and intuitive by its own nature, requires that the entire state of the system is transferred and collected at a single point (the coordinating authority) and that the failure of the coordinator will cause the system to fail. Multi-agent approaches do not suffer from the single point of failure problem, but they significantly increase the complexity of the programming model and introduce uncertainty in the behavior of the system (e.g. in swarm systems [60]).

The **PIM (Process Integrated Mechanism)** is a novel programming

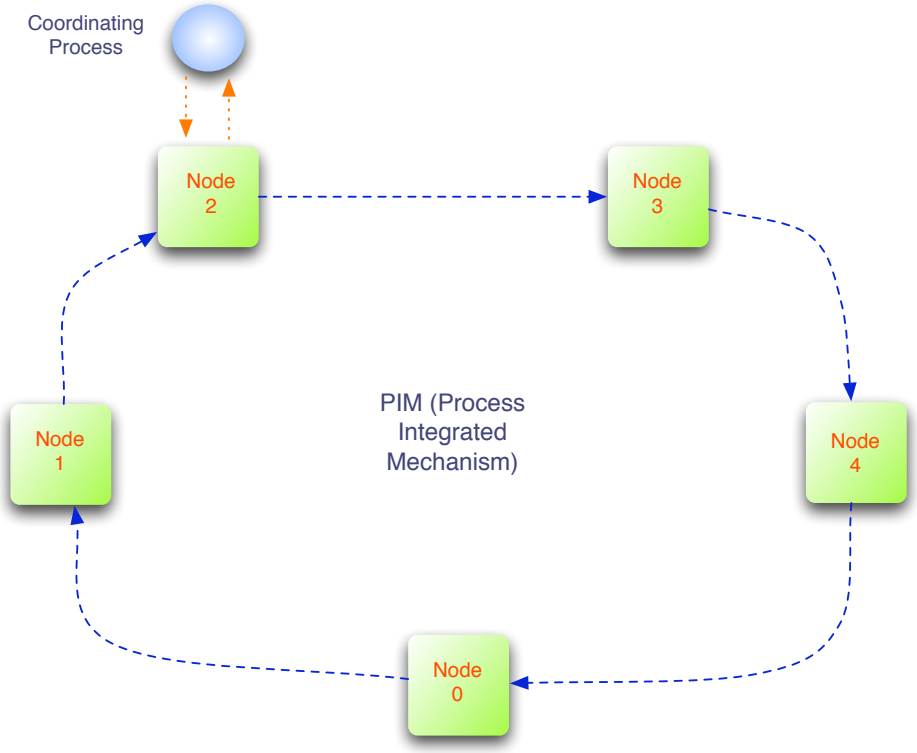


Figure 7.1: A graphical representation of the PIM model

model and a runtime architecture that addresses the distributed coordination problem. The PIM approach retains the advantage of a single coordinating authority while avoiding the structural difficulties that have traditionally led to its rejection in complex settings. The components in the PIM architecture are conceived as parts of a single mechanism, even when they are physically separated and operate asynchronously. The core idea is to retain the perspective of the single controlling authority but abandon the notion that it must have a fixed location within the system. Instead *the*

computational state of the coordinating process is rapidly moved among the component parts of the PIM.
The PIM model (see figure 7.1) consists of a single *Coordinating Process (CP)* and a set of *Components/Nodes* each capable of running the CP. The CP cycles amongst the components at a speed that is sufficient to meet the overall coordination needs of the PIM (e.g. so the robots can coordinate their reaction to new events in a timely manner). The time that the CP runs on a component is called its *Residency Time*. Each component maintains the code for the CP, so the controlling process can move from component to component by passing only a small run-time state using the mechanisms of **strong mobility**. The underlying PIM runtime system manages the actual movement of the CP across the components, and presents the programmer with a virtual machine in which there is a single coordinating process operating with a unified global view and the data remains distributed across the components.
The PIM model addresses the problems associated with traditional approaches. In comparison to centralized approaches, the PIM model ameliorates the robustness problem because the coordinating process is not resident (except transitorily, with backups at all other nodes) at any location. It also ameliorates the communication bottleneck by moving the process to the data rather than the data to the process. While agent-based approaches also address the robustness issue, they introduce significant uncertainty and complexity. The PIM approach retains the conceptual simplicity and ease of programming of the centralized model as it removes the complications of negotiation protocols and timing problems in coordinating multiple autonomous systems.

7.1.1 The PIM as an inverse time-sharing

The basic engineering technique of the PIM derives from time-sharing systems and mobile code technology, and can be in fact described as "inverse time sharing". Time sharing models revolutionized computing because they allowed multiple processes to run on the same computer at the same time as though each was the only process running on that machine. The programmer could construct the program with no concern for the details of the process switching that is actually happening underneath. To the programmer it is as though her program has the entire processor for its own use, even though in reality it is only running in bursts as it is switched in and out.

The PIM model, on the contrary, provides the reverse. To the programmer it still appears that there is one program controlling all the components, but the CP is actually cycling from component to component. Even further, it is as though the memory and data available on each processor is also al-

ways available, as in a distributed memory system. In other words, the set of components appear to be a single entity (i.e., a PIM). The programmer needs not be concerned with the details of the process moving among the processors. In the time sharing model, it is important that each process is run frequently enough to preserve the illusion that it is constantly running on the machine. Likewise, with the PIM model, it is important that the CP runs on each component frequently enough to react appropriately to any changing circumstances in the environment.

7.1.2 Tradeoffs in designing PIMs

There is a tradeoff between the reactivity of the PIM and the amount of computation it can do. A longer residency time reduces the total fraction of time lost to transmission delays, thereby increasing the computational resources available to the CP algorithms. This increases the latency of the CP as it moves amongst the components, thereby decreasing the coordination and reactivity of the PIM. Conversely, a shorter residency time enhances the system's ability to coordinate overall responses to new and unexpected events since the overall cycle time of the CP will be shorter. But as the residency time is reduced, the ratio of the overhead associated with moving the CP increases and thus the computation available to the CP for problem solving decreases. In the extreme case, this could lead to a new form of **thrashing**, where only little computation relevant to coordination is possible because all cycles are being used to move the CP from one component to the next.
These competing factors can be characterized by the following formulas:

```
Cycle-time = number-of-components * (Residency-Time +
          Time-to-move-Coordinating-Process)

Maximum-Percent-useful-computation = Residency-Time /
 (Residency-Time + Time-to-move-Coordinating-Process)
```

A key requirement of the PIM model is that the time taken to cycle the CP among the components is small compared to the reaction time needed by the system. Conditions under which this assumption might fail include situations involving limited bandwidth between components (such as under water) or where remote communication fails altogether, but these conditions will pose difficulties for any distributed system architecture. Note that if the perception and response to an event is wholly local to one component, then the component may react appropriately independent of the coordinating process, much like "reflex" actions in animals. But, responses that require coordinating multiple components are sensitive to the cycle time. A PIM that has a very rapid cycle time can realize highly coordinated behavior. A PIM that has a relatively slow cycle time has more cycles for reflective thinking at the expense of component coordination.

7.1.3 Robustness to Failure

One of the key advantages of PIMs is that they can be robust in the face of component failures with little effort required from the programmer. To attain robustness to component failures, PIM programs must be written in terms of *component capabilities, rather than specific components.* This way, if a component is disabled, then another component with similar capabilities can take its place. Consider the two most common cases where a component becomes disabled or loses communication capability to the other components. The situations differ only in whether the CP is resident on the component at the time it is disabled. In the case where a component that does not have the CP resident disappears, the PIM runtime will detect that the component is missing at the time it attempts to move the CP to it. In this case, the CP is forwarded on to the next component in line in the cycle and the list of available components is updated. The process is shown in

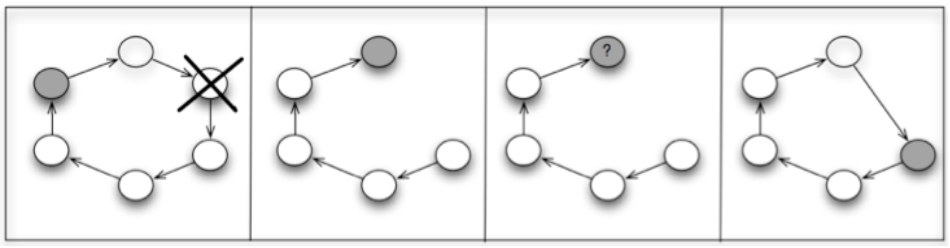


Figure 7.2: Recovery from the loss of a component in the PIM

figure 7.2.

The case in which a component is destroyed while the CP is resident is slightly more complicated. The CP is lost and will not be passed on. It is fairly simple to have a time-out mechanism in the runtime so that this situation is detected. In that case, the runtime then restarts using a copy of the CP from that last known active component. As the CP migration restarts, the PIM continues as before. Because of the short cycle time of the process, the revived CP is only slightly out of date and the PIM continues probably with only minor effect, except that whatever happened on the component that was destroyed it is now missing. This process is shown in figure 7.3. There are more complicated cases of failure as well.

Furthermore, note that the PIM runtime also needs a mechanism for a newly introduced component to join the PIM. By handling these details at the runtime level, the programming of the CP gains significant robustness with minimal programming effort.

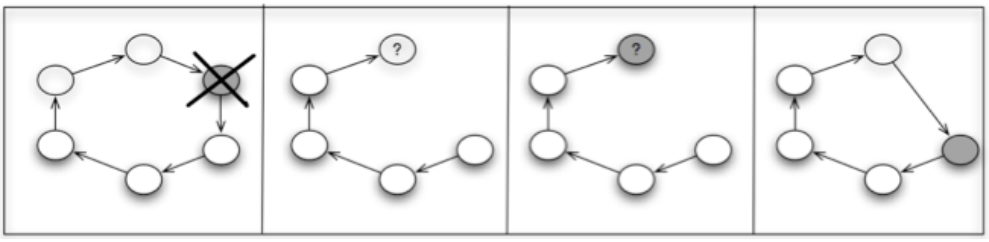


Figure 7.3: Recovery from the loss of the Coordinating Process component in the PIM

7.1.4 Maintaining a Global View

One of the key advantages of PIM models is that the CP can be programmed assuming a *global view*, yet we do not require the communication overhead of maintaining the global view on a single processor. One way to achieve this is to treat the global view as a distributed memory. The global view is distributed amongst the components, where much of the information resides on the component that can compute that information using local processing. Rather than moving the data to the process as it happens in a distributed memory system however, in a PIM, the process moves to the data. In the most naive implementation of this, when the CP needs to access a memory location `X`, it blocks until it is resident on the component that stores `X`. Information that is computed from several disjoint local sources may be stored on any component and accessed in the same way. There are other ways that could optimize the behavior of such a system. One approach is to cache key information and move this with the CP so that it is always available. There is obviously a tradeoff, between the cache size and the overhead in migrating the CP, that requires investigation. At one extreme, there is no cache and the model is a blocking one as described above. At the other extreme, the entire global view could be cached and moved with the CP. While each of these approaches might be effective in certain specific applications, they are unlikely to be effective across a broad range of applications, and intermediate caching approaches will need to be used.

7.1.5 Empirical Studies

It might seem that it is simply too expensive to be moving the CP rapidly among the components. It turns out, however, that the amount of information that needs to be transferred between components can be quite small. First, all the code for the CP is resident on each component, so only the execution state needs to be transferred. At the minimum this would be the current process stack - the stack in the virtual machine with sufficient information so that the next step in the process can be executed. Beyond

that there is a time-space tradeoff on how much data is transferred with the process.

AromaVM	**Payolad Size**	**Avg Cycle Time**	**Overhead per hop**
	0	375.78	25.26
	1024	378.83	26.28
	10240	443.77	47.92
	20480	515.35	71.78
Mobile JikesRVM	**Payolad Size**	**Avg Cycle Time**	**Overhead per hop**
	0	330.85	10.28
	1024	334.95	11.65
	10240	381.47	27.16
	20480	446.76	48.92

Table 7.1: Migration Performance on Aroma and Mobile JikesRVM

To evaluate the feasibility, two prototype implementations of the PIM runtime have been developed. Both support Java as the programming language for the CP. The first implementation uses the Aroma Virtual Machine [72] and the second version uses Mobile JikesRVM. The overhead of passing the CP with varying memory "payloads" has been measured. Three laptops were connected using an 802.11b based ad-hoc network operating at 11 Mbps. The residency time in this test was set to 100 ms. The experiment measured the round trip times for payloads of size 0, 1024, 10240, and 20480 bytes (see table 7.1).

As it can be seen, the migration time for both the virtual machines is close to linear in the size of the CP. This performance can be significantly improved with further development.

7.1.6 Comparison with Agent-based Systems

In order to compare PIM models with agent-based approaches, PIM-based and agent-based teams of robots were built to play a version of "capture the flag" [33]. In this game, each team of robots tries to knock over the opponent's flag by running into it, while defending their own flag. The game ends when one of the team knocks down the opponent's flag or the time expires. Figure 7.4 shows a screen snapshot of the **MobileSim simulator** with five robots on each side playing the game. Figure 7.5 shows the game being played by physical robots - two on each side (Mobile Robots Pioneer 3). The agent-based system is implemented using the **A-globe multi-agent platform** [64]. A-globe is a lightweight multi-agent platform that supports mobility and integration with powerful environment simulation components. A-globe has been used successfully for free-flight collision avoidance among autonomous aerial vehicles and also for modeling collaborative underwater

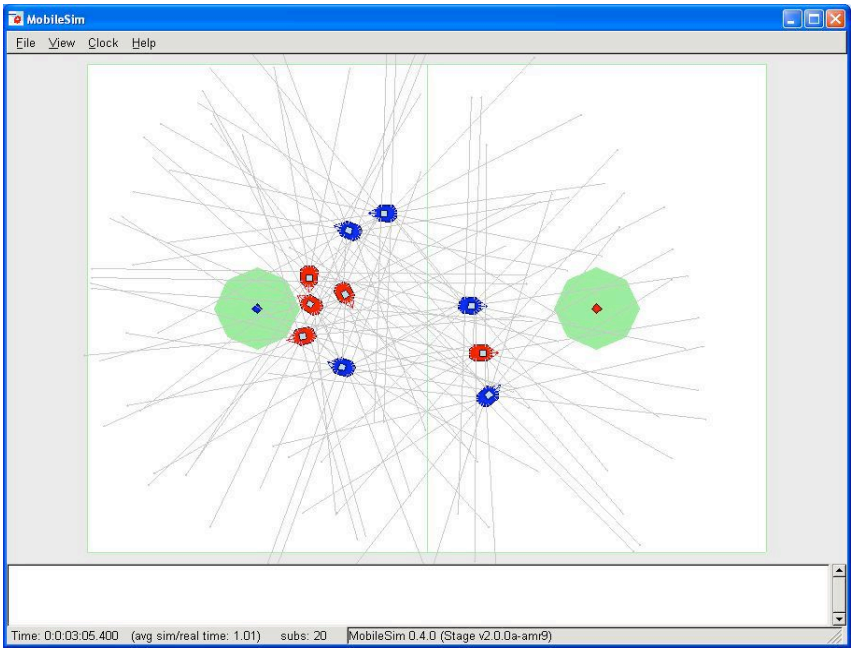


Figure 7.4: Capture The Flag game in the simulator

Figure 7.5: Four Robots play the Capture The Flag game in a parking lot

mine-sweeping search. The environment simulation components support realistic testing of the multi-agent algorithms and facilitate straightforward migration to real distributed environments (such as robotic hardware).

There are several modes to implement multi-agent operations that could be used for comparison with the PIM models:

1. Agents have complete, identical sets of data and attempt to optimize behavior of the entire community. Possible conflicts between agents require negotiation.

2. Agents perform only local decision making on top of complete, identical sets of data. Coordination is achieved by negotiation.

3. Agents perform only local decision making on top of partial data that are available to each individual agent.

It is likely that the PIM model will be effective in each of these modes. Some first experiments, however, compare programming complexity and behavioral efficiency of PIM and A-globe in the mode 1. This mode allowed testing the two approaches using identical decision making strategies. The results on experiments for agent-based systems using mode 2 are reported in [33]. The implemented architecture supports experiments either on real robots or in a simulated environment. Even in the simulated case, the operation of the PIM and A-globe are still real - that is, there are five computers on each side, one for each robot. In the case of the PIM, the CP actually moves between the five components. The only aspect that is simulated is the robots and the physical world.

Some restrictions to make the experiments feasible. First, each team member provides its position to a *position server*, which then distributes the position information to a node using a virtual sensor. The virtual sensor is configured with a sensing range, which is used to determine the visibility of other robots to any given robot. This simplification reduces the impact of sensor inaccuracy on the comparison of the systems. Second, the size of the field was chosen according to the robot's size and number of team members. Robots are not allowed to touch each other, and were designed to stop 10 cm before any obstacle. In 30 runs in the simulation robot environment, the PIM team won 17, the MAS team won 5, and there were 8 ties.

Code Complexity Comparison

One of the main claims for the PIM approach is that it simplifies writing the code. Code complexity can be compared with the simple measure of lines of code. While it is flawed as a measure of complexity, it is still widely used. A second metric is the number of classes. Table 7.2 shows the number of

	Lines (PIM)	Classes (PIM)	Lines (MAS)	Classes (MAS)
Coordination	671	11	4851	33
Role Assignment	538	3	1027	7
Path planning	430	3	681	2

Table 7.2: Comparison of line of codes

lines of code needed to ensure the same behavior in both approaches. Code is divided into three parts:

1. coordination of robots and their actions;
2. optimal role assignment and related strategies;
3. planning collision-free optimal paths.

	Instruction count	Number of classes	Number of methods	Max CC	Methods CC greater than 10
PIM	2327	14	153	8	0
MAS	13428	40	337	55	13

Table 7.3: Comparison of cyclomatic code complexity

In addition, a cyclomatic complexity measure has been used [49]. **Cyclomatic complexity** (CC) is based on graph theory. In a strongly connected graph, the cyclomatic number is equal to the maximum number of linear independent circuits. A program can be associated with a directed graph as follows. Each node in the graph corresponds to a block of code where a control flow is sequential and the arcs correspond to branches taken in the program. The cyclomatic number is equal to total number of possible paths through the program in this case.
The software package CyVis2 [3] was used for computing the cyclomatic complexity. Methods with CC less than five are considered easy to understand and debug. Widely recommended maximum value of CC is 10. Methods with a CC greater than 10 are hard to keep in mind as a whole and therefore are hard to understand and debug. Results in table 7.3 show comparison of code length in instruction (bytecode) count and also total number of classes and methods needed for coordination and role assignment. The table also shows the maximum CC for any method, and the number of methods with a CC greater than 10. Parts of the code responsible for path planning are excluded from this comparison as it is hard to compare code written in C++ with code written in Java.
It can be clearly seen that the length of code as well as code complexity needed by the MAS approach to ensure the same coordinated behavior is

much larger than the PIM approach. The key cause is that the PIM allows an effective centralized coordination, using all the knowledge about the members of the group and coordinating them at once. In contrast, the MAS approach needs to implement negotiation protocols, knowledge sharing, and synchronization mechanisms.

7.2 Agile Computing

The other project where Mobile JikesRVM has been successfully integrated has been the Agile Computing service-oriented architecture. **Service oriented architectures (SoAs)** are a popular approach to designing and building networked and distributed systems. SoAs allow the realization of complex distributed applications through the composition of services according to the definition of a specific business process. This offers the opportunity for both an extensive service reuse and the integration of heterogeneous services, with significant savings in distributed applications development costs and time.

Most SoAs were designed to operate on either corporate networks or in the Internet environment, with the purpose of separating business processes and rules from the implementation of basic service functions. Hence, traditional SoA implementations are based on centralized service directories and make strong assumptions about relatively fixed network topologies, large bandwidth availability, and high network stability.
However, a growing interest in running SoAs in MANET environments has recently emerged. In fact, SoAs allow the dynamic (re)composition of services at run-time, thus enabling the ad-hoc realization of complex distributed applications. In addition, SoA-based applications build on top of lean and modular services, which are better suited for the deployment on mobile and resource-constrained nodes than traditional heavyweight services. Finally, the modular architecture of SoA-based applications allows the dynamic replacement of services, therefore enabling application adaptation to the constantly changing network topology of MANETs.

Unfortunately, the realization of SoAs in MANETs is a very challenging task which requires a significant re-engineering of existing SoA platforms. In particular, the MANET environment is not well suited for the deployment of traditional centralized service broker(s). This calls for a decentralized and distributed peer-to-peer approach to service discovery, which can efficiently find service instances and select the best suited ones for exploitation, e.g. those satisfying topological proximity and/or reliability constraints. Also, frequent variations in network topology and resource availability introduce the opportunity for **dynamic service migration**, in order to improve the

performance and availability of subscribed services. Finally, in MANET environments, bandwidth is typically very scarce, thus imposing strict efficiency constraints on signaling protocols involved in service discovery, migration, and invocation.
There is a growing effort from both the industry and the academia to develop ad-hoc solutions for SoA-based service discovery and migration in MANETs. However, it is still unclear whether these proposals enable the SoA architectural style to deliver satisfying performance levels in MANETs. As a result, there is the need to gather some insights on the performance and efficiency of state-of-the-art solutions.

In this context, another part of the work done at IHMC has been implementing service migration with Mobile JikesRVM and then doing an extensive experimental evaluation of SoA-based applications built on top of the Agile Computing middleware. The Agile Computing middleware is an IHMC solution for the realization of SoAs designed to operate in MANETs. It is based on a peer-to-peer architecture and an adaptive and opportunistic paradigm for resource/service discovery and exploitation called **agile computing**. In particular, the research work reported here focuses on the dynamic service migration functions of the Agile Computing middleware, made possible by exploiting thread migration facilities from Mobile JikesRVM. Service migration in the Agile Computing middleware enables load-balancing, fault-tolerance, and the application of autonomic computing principles to SoAs.

7.2.1 The Agile Computing Middleware

This section provides a short overview of the notion of agile computing and of the Agile Computing middleware. *Agile computing* is a novel metaphor for distributed and networked systems. It emphasizes designing systems to be opportunistic in discovering and exploiting resources in a dynamic environment as well as being able to quickly adapt to changes in such an environment. The word agile is used to highlight both the rapid discovery and exploitation of resources and the ability to take advantage of highly transient resources.

The Agile Computing middleware is a specific implementation of the agile computing metaphor that proposes a peer-to-peer approach for the realization of SoA-based distributed applications in the MANET environment. The middleware consists of six major components:

- the Agile Computing Interface (ACI) Kernel
- the Group Manager

- the Service Manager
- Mockets
- AgServe
- FlexFeed

The ACI Kernel provides container functions for hosting and executing services. The ACI Kernel also instantiates and contains the Group Manager component, which supports dynamic resource and service discovery. The Service Manager provides service matching functions on top of the Group Manager, providing applications with a convenient interface based on XML and XPath. Mockets is a communications library providing several advanced features that were purposely designed for the MANET environment. The endpoint migration capability of Mockets is particularly relevant to the service migration capability described later. AgServe provides support for dynamic deployment, activation, and migration of services. Finally, FlexFeed is a publish-subscribe system that handles hierarchical data dissemination, policy-based transformation of data, and in-stream data processing. More information about agile computing and the Service Manager, Mockets, and FlexFeed is available in [73], [26] and [78] respectively. In the remainder of this section, only components of interest to service migration are described in more details.

Group Manager

The Group Manager is a flexible, application-level component that supports resource and service discovery. It enables the agile and opportunistic exploitation of resources by optimizing queries to find nodes in network proximity and/or nodes that are resource rich or have excess capacity.
The Group Manager supports simultaneously pushing information proactively as well as searching for information reactively. The design of the Group Manager also allows nodes to control the radius of advertisement or search (in terms of the number of hops). The Group Manager API has been designed to be extremely simple and generic, facilitating its use in a wide range of applications. For example, the ACI Kernel uses the Group Manager to propagate node resource information, the Service Manager uses it to publish services in XML and search for services using XPath queries, and FlexFeed uses the Group Manager to find data sources for subscribers.
The Group Manager runs either as part of the ACI Kernel or embedded inside an application, using either UDP broadcast, multicast. Versions of the Group Manager are available for both Java and C++, and operating at the application layer facilitates easy, piecewise integration into existing applications. These and other features combine together to make the Group

Manager well suited to MANET environments. More information about the Group Manager component is available in [76].

AgServe

The AgServe component supports dynamic deployment, activation, and migration of services, thus enabling the opportunistic exploitation of resources. For example, as nodes with free resources become available, service instances might be migrated from heavily loaded servers to other nodes, thereby improving the overall performance of the system (self-optimizing behavior). Service migration also allows the system to react to accidental events, such as a power loss or an incoming attack (survivability behavior).
AgServe extends the ACI Kernel service container to provide transparent service migration. A service running inside the service container can be asynchronously stopped, its execution state captured, moved to a new service container (usually on a different node), and then restarted. This migration is transparent to both the service itself as well as the client utilizing the service. Service migration allows AgServe to react to changes in the environment and is crucial to read the goal of agility.
The ACI Kernel contains two Java-compatible service containers, one based on the Aroma VM and the other based on Mobile JikesRVM, which both support transparent service migration for services implemented in Java. The ACI Kernel builds a resource utilization profile for each service, keeping track of the CPU utilization as well as the bytes sent and received over the network for each invocation. This information, along with the resource availability information from other nodes, is used by the coordinator in the kernel to migrate service instances between nodes. The checkpointing capabilities of Aroma and Mobile JikesRVM, combined with the endpoint migration capability of Mockets, allow service instances to be migrated in a manner completely transparent clients, even during the midst of an invocation operation. More information about AgServe can be found in [75].

Some preliminary experimental results

Some preliminary experiments were conducted to evaluate the performance of the agile computing middleware (and other are being carried out at the time of writing). The experiment presented here evaluates the service migration capability provided by AgServe. It shows the benefits of opportunistically taking advantage of transient resources and measures the overhead of service migration. The scenario is shown in figure 7.6 and consists of a number of client nodes, and a smaller number of servers and opportunistic nodes (i.e., nodes that become available for short periods of time during which their resources are exploited). For this experiment, two client nodes, one server node, and one transient node were utilized. The two client nodes

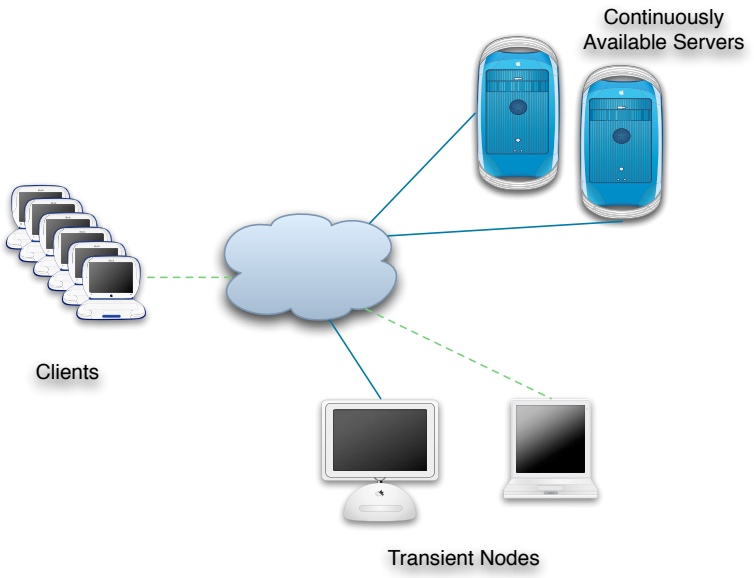


Figure 7.6: Service Migration Scenario

continuously invoke a factorization service that is CPU intensive, thereby overloading the only server that is available. When a transient node becomes available, the middleware transparently migrates one of the service instances to the transient node. When the node is about to become unavailable (e.g., being shutdown), the service is migrated back to the server node. Figure 7.7 below shows the results of the service migration on the performance of the service from the client's perspective. The independent variable in this experiment is the duration of time for which the transient node is available. The dependent variables are the turnaround times for the service invocation from the two clients. The time to execute a service, if there is no other load on the server node, is 54760 ms. If two clients simultaneously execute the service and there is no transient node to exploit, the overall time for both clients to finish is 98570 ms. When the transient node is available for 5 seconds, the overall time actually becomes worse (100171 ms), which is an indication of the overhead of service migration. However, as the duration of the transient node increases, the overall performance improves. The break-even point is around 8 seconds, which is a measure of the overall agility of the system. It is the shortest length of time for which an opportunistic node may be exploited without any performance degradation. If a transient node is available for a longer length of time, the middleware shows a positive performance improvement.

Transient Node	Service #1	Service #2	Overall Time
Time (sec)	(Execution Time in ms)		
0	98362	98570	98570
5	99421	100171	100171
10	96683	97337	97337
15	92019	93443	93443
20	88990	89203	89203
30	79324	81220	81220
40	75322	78208	78208
50	64870	70686	70686
60	58614	66706	66706
70	58218	64826	64826
Baseline time for service execution - 54760 ms			

Figure 7.7: Service Migration Performance

Chapter 8

Conclusion

The present dissertation thesis has focused on a novel thread migration approach for the Java programming language. Mobile JikesRVM is the name of the framework implementing such an approach on top of the IBM Jikes Research Virtual Machine project.

With respect to previous systems, Mobile JikesRVM allows fast and complete thread state capture and restoration, even in the presence of JIT compilation and state-of-the-art code optimizations. Thanks to the introduction of a new IR instruction (called `OSR_Point`), also frames produced by the optimizing JIT compiler are fully observable like baseline ones. Java threads are transparently migrable at specific safe points during their execution, both proactively and reactively. Thread restoration is achieved by means of a specially designed technique, called "self-installing method bytecodes".

Three performance aspects (migration cost, overhead on the JVM and application execution overhead) have been extensively analyzed and evaluated using the DaCapo JVM benchmarks suite. The overhead imposed by Mobile JikesRVM has been kept very limited (from 0.4% to 3%) thanks to the introduction of the Mobile Method concept and to minimally invasive modifications to the original JikesRVM. Capture and restore times are encouraginly fast (in the order of ns and msec respectively), making Mobile JikesRVM a valid framework on which to run distributed Java applications.

Two real distributed applications (robot coordination with the PIM and Agile Computing) using strong thread migration have been successfully implemented in collaboration with IHMC (Institute for Human and Machine Cognition), Pensacola, Florida. Even though some preliminary results of this collaboration have been reported here, the research work on these projects is still actively being carried out and is thus part of future work for the student after his PhD.

Bibliography

[1] *The Aglets Mobile Agent Platform website.* http://aglets.sourceforge.net.

[2] *Compiler optimizations as in Wikipedia.* http://en.wikipedia.org/wiki/Compiler_optimization.

[3] *The CyVis2 project website on sourceforge.* http://cyvis.sourceforge.net/.

[4] *The DaCapo project website.* http://dacapobench.org.

[5] *Institute for Human and Machine Cognition (IHMC), Pensacola, Florida, US.* http://www.ihmc.us.

[6] *The Java Compiler Compiler project website.* https://javacc.dev.java.net/.

[7] *The JikesRVM project site.* http://jikesrvm.sourceforge.net.

[8] *The JToe project web page.* http://jtoe.sourceforge.net.

[9] *The Sumatra project website.* http://www.cs.arizona.edu/projects/sumatra/.

[10] A. Acharya, M. Ranganathan, and J. H. Saltz. Sumatra: A language for resource-aware mobile programs. In Vitek and Tschudin [81], pages 111–130.

[11] F. E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[12] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[13] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. F. Mergen, J. C. Shepherd, and S. E. Smith. Implementing jalapeño in java. In *OOPSLA*, pages 314–324, 1999.

[14] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. McKinley, M. Mergen, J. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building and open-source research community. *IBM Systems Journal*, 44(2), 2005.

[15] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2000.

[16] F. Berman, G. Fox, and A. J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, Inc., New York, NY, USA, 2003.

[17] S. Bouchenak and D. Hagimont. Pickling threads state in the java system. In *TOOLS (33)*, pages 22–32. IEEE Computer Society, 2000.

[18] S. Bouchenak, D. Hagimont, S. Krakowiak, N. D. Palma, and F. Boyer. Experiences implementing efficient java thread serialization, mobility and persistence. *Softw., Pract. Exper.*, 34(4):355–393, 2004.

[19] F. Breg and C. D. Polychronopoulos. Java virtual machine support for object serialization. In *Joint ACM-ISCOPE conference on Java Grande*, pages 173–180, 2001.

[20] R. R. Brooks. Mobile code paradigms and security issues. *IEEE Internet Computing*, May - June 2004.

[21] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[22] D. R. Butenhof. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[23] G. Cabri, L. Ferrari, L. Leonardi, and R. Quitadamo. Strong agent mobility for aglets based on the ibm jikesrvm. In H. Haddad, editor, *SAC*, pages 90–95. ACM, 2006.

[24] G. Cabri, L. Leonardi, and R. Quitadamo. Enabling java mobile computing on the ibm jikes research virtual machine. In Gitzel et al. [37], pages 62–71.

[25] G. Cabri, L. Leonardi, and F. Zambonelli. Weak and strong mobility in mobile agents applications. In *2nd International Conference and Exhibition on The Practical Application of Java*, April 2000.

[26] M. Carvalho, N. Suri, and M. Arguedas. Mobile agent-based communications middleware for data streaming in the battlefield. In *IEEE Military Communications Conference (MILCOM 2005)*, October 2005.

[27] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed application with mobile code paradigms. In R. Taylor, editor, *International Conference on Software Engineering (ICSE)*, pages 22–23. ACM Press, 1997.

[28] A. J. Chakravarti, X. Wang, J. O. Hallstrom, and G. Baumgartner. Implementation of strong mobility for multi-threaded agents in java. In *ICPP*, pages 321–. IEEE Computer Society, 2003.

[29] L. Courtrai, Y. Maheo, and F. Raimbault. Expresso: a library for fast transfer of java objects. In *Myrinet User Group Conference*, 2000.

[30] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. Analyzing mobile code languages. In Vitek and Tschudin [81], pages 93–110.

[31] J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst.*, 2(2):191–202, 1980.

[32] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *CGO*, pages 241–252. IEEE Computer Society, 2003.

[33] K. Ford, N. Suri, K. Kosnar, P. Jisl, P. Benda, M. Pechoucek, and L. Preucil. A game-based approach to comparing different coordination mechanisms. In *IEEE International Conference on Distributed Human-Machine Systems (DHMS)*. IEEE, 2008.

[34] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Trans. Software Eng.*, 24(5):342–361, 1998.

[35] S. Fünfrocken. Transparent migration of java-based mobile agents: Capturing and reestablishing the state of java programs. *Personal and Ubiquitous Computing*, 2(2), 1998.

[36] General Magic. *Telescript Language Reference*, October 1995.

[37] R. Gitzel, M. Aleksy, and M. Schader, editors. *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java, PPPJ 2006, Mannheim, Germany, August 30 - September 1,*

2006, volume 178 of *ACM International Conference Proceeding Series*. ACM, 2006.

[38] R. S. Gray, G. Cybenko, D. Kotz, R. A. Peterson, and D. Rus. D'agents: Applications and performance of a mobile-agent system. *Software-Practice and Experience*, 32(6), May 2002.

[39] C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? *Lecture Notes in Computer Science*, 1222:22–57, April 1997.

[40] O. Holland and C. Melhuis. Stigmergy, self-organization and sorting in collective robotics. In *Artificial Life*, volume 5, pages 173–202, 1999.

[41] T. Illmann, T. Krueger, F. Kargl, and M. Weber. Transparent migration of mobile agents using the java platform debugger architecture. In G. P. Picco, editor, *Mobile Agents*, volume 2240 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2001.

[42] N. R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.

[43] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[44] D. Kotz, R. S. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, 1997.

[45] D. Kotz and F. Mattern, editors. *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zürch, Switzerland, September 13-15, 2000, Proceedings*, volume 1882 of *Lecture Notes in Computer Science*. Springer, 2000.

[46] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3), 1999.

[47] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, 1999.

[48] A. Malgi, N. Bansod, and B. K. Choi. String: Efficient implementation of strongly migrating mobile agents in java. In M. J. Oudshoorn and S. Rajasekaran, editors, *ISCA PDCS*, pages 314–321. ISCA, 2005.

[49] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[50] S. Meloan. *The Java HotSpot Performance Engine: An In-Depth Look*. Sun Microsystems, http://java.sun.com/developer/technicalArticles/Networking/HotSpot/index.html, 1999.

[51] S. Oaks and H. Wong. *Java Threads*. O'Reilly Media, Inc., 2004.

[52] OMG (Object Management Group). *CORBA: Architecture and Specification*, August 1995.

[53] J. K. Ousterhout. Tcl: An embeddable command language. In *USENIX Winter*, pages 133–146, 1990.

[54] D. A. Park and S. V. Rice. A framework for unified resource management in java. In Gitzel et al. [37], pages 113–122.

[55] H. Peine and T. Stolpmann. The architecture of the ara platform for mobile agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Mobile Agents*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61. Springer, 1997.

[56] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[57] A. J. Project. *The Byte Code Engineering Library*. Apache Foundation, http://bcel.sourceforge.net/.

[58] R. Quitadamo, G. Cabri, and L. Leonardi. Mobile jikesrvm: A framework to support transparent java thread migration. *Science of Computer Programming, Elsevier*, 70(2+3):221–240, February 2008.

[59] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[60] E. Sahin. Swarm robotics: From sources of inspiration to domains of application. In E. Sahin and W. M. Spears, editors, *Lecture Notes in Computer Science*, volume 3342, pages 10–20. Springer, 2004.

[61] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in java. In Kotz and Mattern [45], pages 16–28.

[62] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In P. Ciancarini and A. L. Wolf, editors, *COORDINATION*, volume 1594 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 1999.

[63] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 21–36, Canterbury, UK, 1997. Chapman and Hall, London.

[64] D. Sislak, M. Rehak, M. Pechoucek, M. Rollo, and D. Pavlicek. A-globe: Agent development platform with inaccessibility and mobility support. In M. K. Rainer Unland and M. Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 21–46. Birkhauser Verlag, 2005.

[65] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(2):277–296, October 1990.

[66] Standard Performance Evaluation Corporation (SPEC), http://www.spec.org/jvm98/. *SPECjvm98 Documentation*, 1.04 edition, February 2001.

[67] Standard Performance Evaluation Corporation (SPEC), http://www.spec.org/jbb2000. *SPECjbb2000 Documentation*, 1.02 edition, September 2003.

[68] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers. In *SOSP*, pages 68–78, 1995.

[69] T. Suezawa. Persistent execution state of a java virtual machine. In *Java Grande*, pages 160–167, 2000.

[70] Sun Microsystems, http://java.sun.com/javase/technologies/core/toolsapis/jpda/. *The Java Platform Debugger Architecture*.

[71] Sun Microsystems. *The Java Language Specification*, October 1995.

[72] N. Suri. State capture and resource control for java: The design and implementation of the aroma virtual machine. In *Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[73] N. Suri, J. Bradshaw, M. Carvalho, T. Cowin, M. Breedy, P. Groth, and R. Saavedra. Agile computing: Bridging the gap between grid computing and ad-hoc peer-to-peer resource sharing. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[74] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. Nomads: toward a strong and safe mobile agent system. In *Agents*, pages 163–164, 2000.

[75] N. Suri, M. Rebeschini, M. Arguedas, M. Carvalho, S. Stabellini, and M. Breedy. Towards an agile computing approach to dynamic and adaptive service-oriented architectures. In *First IEEE Workshop on Autonomic Communication and Network Management (ACNM'07)*, 2007.

[76] N. Suri, M. Rebeschini, M. Breedy, M. Carvalho, and M. Arguedas. Resource and service discovery in wireless ad-hoc networks with agile computing. In *IEEE Military Communications Conference (MILCOM 2006)*, October 2006.

[77] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2001.

[78] M. Tortonesi, C. Stefanelli, N. Suri, M. Arguedas, and M. Breedy. Mockets: A novel message-oriented communication middleware for the wireless internet. In *International Conference on Wireless Information Networks and Systems (WINSYS 2006)*, August 2006.

[79] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in java. In Kotz and Mattern [45], pages 29–43.

[80] G. Vigna. Mobile agents: Ten reasons for failure. In *Mobile Data Management*, pages 298–299. IEEE Computer Society, 2004.

[81] J. Vitek and C. F. Tschudin, editors. *Mobile Object Systems - Towards the Programmable Internet, Second International Workshop, MOS'96, Linz, Austria, July 8-9, 1996, Selected Presentations and Invited Papers*, volume 1222 of *Lecture Notes in Computer Science*. Springer, 1997.

[82] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *IWMM*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer, 1992.

[83] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *CLUSTER*, pages 381–388. IEEE Computer Society, 2002.